# CÓDIGO PARA ARTISTAS

MIGUEL OLIVEIRA

uma abordagem simples e descomplicada à algoritmia e programação para públicos não convencionais

MIGUEL OLIVEIRA

# CÓDIGO PARA ARTISTAS

uma abordagem simples e descomplicada à algoritmia e programação para públicos não convencionais

Direitos de autor © Miguel Oliveira, 2026

Autor: Miguel Oliveira
Editor: Miguel Oliveira
Título: Código para artistas
Sub-Título: Uma abordagem simples e descomplicada à algoritmia e programação para públicos não convencionais
Capa: Augusto Coelho e Miriam Reis
Organização e paginação: Miriam Reis
Local de edição: Escola Superior de Design, Gestão e Tecnologias da Produção de Aveiro-Norte (ESAN), Universidade de Aveiro

ISBN
978-989-33-9397-0

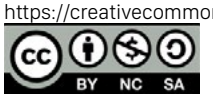
Depósito Legal
561580/26

Todos os direitos reservados. Nenhuma parte desta publicação pode ser reproduzida, armazenada em sistema de recuperação ou transmitida sob qualquer forma ou por qualquer meio — eletrónico, mecânico, fotocópia, gravação ou outro — sem autorização prévia do autor.

Licença
Este trabalho está licenciado sob uma Licença Creative Commons Atribuição–NãoComercial–Partilhalgual 4.0 Internacional (CC BY-NC-SA 4.0).

Pode ser partilhado e adaptado para fins não comerciais, desde que seja atribuída a autoria e que as obras derivadas sejam distribuídas sob a mesma licença.

https://creativecommons.org/licenses/by-nc-sa/4.0/



# Agradecimentos

Este livro nasceu no contexto das aulas da disciplina Sistemas de Informação, da Licenciatura em Design de Produto e Tecnologia (Escola Superior Aveiro-Norte da Universidade de Aveiro) onde venho explorando, com os meus estudantes, formas de aproximar o pensamento algorítmico das artes e das humanidades.

A todos eles devo um agradecimento especial: muitas das explicações, exemplos e exercícios aqui incluídos resultam diretamente das perguntas, dificuldades e curiosidades que surgiram em sala de aula. Ensinar é sempre também aprender, e este livro é, em grande parte, fruto desse processo partilhado.

Agradeço igualmente aos colegas que, de diferentes áreas, incentivaram este diálogo entre programação e humanidades, mostrando que as fronteiras entre disciplinas são mais permeáveis do que por vezes imaginamos. As conversas informais, as sugestões de leitura e as críticas construtivas foram fundamentais para amadurecer várias das ideias aqui apresentadas.

Uma palavra de reconhecimento também para todos os autores e comunidades que defendem o acesso aberto ao conhecimento e que desenvolvem ferramentas, linguagens e recursos pedagógicos que tornam possível ensinar programação de forma mais acessível e interdisciplinar.

Um agradecimento especial à Professora Rita Simões pela sua ajuda na fase embrionária deste projeto. Um agradecimento especial aos Designers e docentes Augusto Coelho e Miriam Reis que, sem hesitar, aceitaram participar na elaboração gráfica deste trabalho.

Por fim, agradeço à minha família pelo apoio, paciência e encorajamento ao longo do tempo em que este livro foi sendo escrito e revisto. Sem esse suporte silencioso, muitos projetos intelectuais simplesmente não chegariam ao fim. Qualquer erro ou limitação que permaneça neste trabalho é, naturalmente, da minha inteira responsabilidade.

# Prefácios

Como CEO da Naoris Protocol, e ao longo de uma carreira dedicada à inovação tecnológica, sempre acreditei que o verdadeiro avanço surge quando a criatividade humana e a precisão computacional caminham lado a lado. Este livro, Código para Artistas, não é apenas uma introdução à programação: é uma aposta estratégica no futuro das indústrias criativas e no desenvolvimento de talentos híbridos que Portugal e o mundo tanto precisam.

Pessoalmente, vejo com clareza como o mercado está a evoluir. Os designers, escritores e artistas que dominam algoritmos e Python deixarão de ser apenas consumidores de tecnologia para se tornarem os seus principais criadores e líderes. Num mundo cada vez mais digital, esta capacidade torna-se uma vantagem competitiva decisiva, e permite transformar ideias em produtos inovadores, otimizar processos criativos e liderar projetos que misturam arte com escala tecnológica. Acredito que estamos a formar não só programadores, mas visionários capazes de redefinir a economia criativa.

É com orgulho pessoal e convicção estratégica que a Naoris Protocol apoia esta obra. Que ela inspire muitos a darem o salto: unir a sua imaginação à potência do código. O futuro não pertence aos especialistas isolados, mas aos criadores completos. Este livro é o primeiro passo. **— David Carvalho, CEO da Naoris Protocol**

---

Código para Artistas é uma contribuição notavelmente pensada para a educação em tecnologia. O que torna este livro especial é o seu genuíno respeito pelos públicos não técnicos, frequentemente ignorados pelos manuais tradicionais de programação. Em vez de simplificar o material, os autores apresentaram os algoritmos e a programação em Python através de uma perspetiva que ressoa com mentes criativas, recorrendo a analogias familiares como comparar um algoritmo a uma receita de bolo. Este tom acessível, mas rigoroso, demonstra uma profunda apreciação pela inteligência dos leitores e uma vontade sincera de democratizar o pensamento computacional.

O livro atinge um equilíbrio impressionante entre acessibilidade e profundidade. Avança de forma metódica desde os conceitos fundamentais como o que é um algoritmo, o funcionamento das variáveis e as estruturas básicas de controlo. Até aos ciclos, listas, funções e desafios matemáticos clássicos, como a série de Fibonacci e a lenda do tabuleiro de xadrez.

Numa época em que a intersecção entre arte e tecnologia ganha centralidade nas indústrias criativas, este livro preenche uma lacuna fundamental. Demonstra de forma convincente que a programação não é um território exclusivo dos engenheiros, mas uma ferramenta versátil que amplia as possibilidades criativas. Ao formar uma nova geração de profissionais híbridos, à vontade tanto no mundo técnico como no artístico, o Código para Artistas contribui para algo bem maior do que a literacia em programação: fomenta uma cultura de inovação, experimentação e aprendizagem ao longo da vida em todas as áreas do conhecimento. **— Sumit Chauhan, CTO da Naoris Protocol***

# Introdução

**Porquê aprender algoritmos e programação?**
Aprender algoritmos e programação pode parecer uma tarefa destinada apenas a cientistas da computação ou engenheiros, mas a verdade é que essas habilidades são extremamente valiosas para pessoas de todas as áreas, incluindo design, letras e artes. Ao longo deste livro, vamos explorar o porquê da sua importância.

Em primeiro lugar, a compreensão de algoritmos ajuda a desenvolver o pensamento lógico e a resolução de problemas. Os algoritmos são, basicamente, instruções passo a passo para resolver problemas específicos. Ao aprender a criar e entender algoritmos, irá aprimorar a sua capacidade de decompor problemas complexos em partes menores, facilitando a sua resolução e gestão. Isso é útil não apenas na programação, mas em qualquer campo que envolva a resolução de problemas, como o design de um novo produto, a criação de uma narrativa coerente ou a organização de uma exposição de arte.

Neste livro, iremos abordar a linguagem de programação Python. É uma linguagem de programação acessível e bastante poderosa que pode ser usada para uma ampla gama de aplicações. Para designers, Python pode automatizar tarefas repetitivas, como redimensionar imagens ou gerar padrões complexos. Para escritores e artistas, Python pode ser usado para analisar grandes volumes de texto, identificar padrões e até mesmo criar conteúdos artísticos.

A matemática, que é a base de muitos algoritmos, também desempenha um papel crucial aqui. Por exemplo, conceitos matemáticos como álgebra linear e geometria são fundamentais para gráficos computacionais e animação, áreas de grande interesse para designers e artistas.

Além disso, a interseção entre arte e tecnologia torna-se cada vez mais relevante. Conhecimentos de programação e algoritmos permitem que artistas e designers criem obras interativas e inovadoras. Imagine um designer de moda que usa algoritmos para criar padrões únicos de tecido ou um artista visual que utiliza Python para gerar arte digital baseada em dados. Essas habilidades não só ampliam as possibilidades criativas, mas também aumentam a competitividade no mercado de trabalho.

Por fim, aprender algoritmos e Python promove uma mentalidade de aprendizagem contínua. Com a tecnologia em constante evolução, aqueles que têm a capacidade de aprender novas ferramentas e técnicas, rapidamente estarão na vanguarda da criatividade, do mercado de trabalho e da organização eficiente de tarefas ou projetos. Para alunos de design, letras e artes, isso significa estar preparado para integrar novas tecnologias no seu trabalho e explorar novas formas de expressão.

Resumindo, a aprendizagem de algoritmos e Python, suportada pela matemática, não é apenas para programadores. É uma habilidade valiosa que pode enriquecer o trabalho de designers, escritores e artistas, proporcionando novas ferramentas para a criatividade e a inovação.

**Porquê mais um livro sobre algoritmos e programação?**
Publicar um livro sobre algoritmos e Python, especialmente para públicos não convencionais como alunos de design, letras e artes, é um desafio que pretende desmistificar estas temáticas. Tem por objetivo servir como recurso educativo essencial, preenchendo uma lacuna significativa nos materiais didáticos existentes e altamente

técnicos. Muitos destes materiais sobre algoritmos e programação são orientados para estudantes de ciências exatas, deixando de parte aqueles que poderiam beneficiar (de forma considerável) dessas capacidades em outras áreas. Este livro aborda esses temas de maneira acessível e relevante para designers, escritores e artistas, desejando democratizar o conhecimento e tornar a tecnologia mais inclusiva.

Além disso, os assuntos podem inspirar e motivar leitores a explorar novas fronteiras criativas. Ao mostrar exemplos práticos de como algoritmos e Python, podem ser aplicados em projetos de design, literatura e arte, pode despertar o interesse e a curiosidade dos leitores. Isso não só amplia o horizonte de possibilidades criativas, mas também encoraja uma mentalidade de inovação e experimentação. Por exemplo, um designer gráfico pode descobrir novas técnicas para criar padrões complexos, enquanto um escritor pode aprender a analisar grandes volumes de texto para encontrar novas narrativas.

Outro ponto importante é que livro pode ajudar a desmistificar a programação e torná-la mais acessível. Muitas pessoas ainda veem a programação como uma competência difícil e exclusiva para *nerds*. No entanto, ao apresentar esses conceitos de maneira clara e relacionável, o livro mostra que qualquer pessoa, independentemente de sua formação, pode aprender a programar e usar algoritmos para resolver problemas criativos, dotando o leitor de novas competências, dando-lhe a confiança necessária para explorar novas ferramentas e técnicas.

Ainda, o livro pode contribuir para a formação de uma nova geração de profissionais híbridos, que combinam habilidades técnicas e artísticas. Num mundo cada vez mais digital e interconectado, a capacidade de integrar conhecimentos de diferentes áreas é uma vantagem competitiva significativa. Profissionais que dominam tanto a programação quanto as artes podem liderar projetos inovadores, criar soluções únicas e destacarem-se num mercado de trabalho em constante evolução.

Em resumo, escrever um livro sobre a importância de aprender algoritmos e Python para públicos não convencionais é uma oportunidade de educar, inspirar e enriquecer. É uma maneira de tornar a tecnologia mais acessível e de promover uma cultura de inovação e criatividade em todas as áreas do conhecimento.

**Porquê utilizar exercícios de matemática para aprender algoritmos e programação?**
Este livro apresenta uma coleção de problemas matemáticos simples que servem como uma abordagem eficaz para o ensino e a prática de programação. A escolha desses problemas não é aleatória, mas sim motivada por vários benefícios que eles oferecem. Em primeiro lugar, ajudam a desenvolver o raciocínio lógico e analítico, essenciais para qualquer programador. Além disso, permitem a estruturação de questões relacionadas a tópicos como álgebra e geometria, promovendo a habilidade de decompor problemas complexos em partes menores, uma prática fundamental na construção de algoritmos. Outro aspeto importante é o incentivo à eficiência e precisão na resolução de problemas, com a modelação rigorosa que pode ser diretamente traduzida em código. A capacidade de abstração também é trabalhada, uma vez que os exercícios incentivam o programador a focar nos aspetos essenciais dos problemas, descartando detalhes irrelevantes, o que é crucial para uma programação eficiente. Assim, os exercícios matemáticos e a programação, por estarem profundamente interligados, tornam-se uma ferramenta valiosa para a aprendizagem de algoritmos e programação. Por fim, esperamos também contribuir para desmistificar a ideia de que a matemática é um "papão", mostrando-a como algo acessível e útil no contexto da programação.
Sem mais assunto, vamos passar à ação! Preparado?

# INDICE


**Introdução** **5**

**Linguagem Python e a sua interface de desenvolvimento** **9**

Porquê programar em Python? 9

IDLE – Integrated and Development Learning Enviroment 9

**Introdução à algoritmia** **11**

O que é um algoritmo? 11

A receita de um bolo ou a representação simples de um algoritmo 12

Construção de algoritmos 13

Passos na construção de algoritmos 14

Características fundamentais de um algoritmo 14

Formas de representação de algoritmos 15

O clássico ***Hello World!*** ***16***

**Breves conceitos de introdução à programação** **18**

Categorias de linguagens de programação 18

Termos comuns na programação 19

***Debugging*** – Deteção de erros 20

Variáveis 21

***Input*** (entrada de dados) 24

Output (saída de dados) 24

***Inputs*** e ***Outputs*** (exemplos) 25

Operadores em programação 27

Operadores aritméticos em programação 27

Operadores de atribuição em programação 28

Operadores de comparação em programação 29

Operadores lógicos em programação 31

**Estruturas de controlo em programação** **32**

Sequência 32

Seleção 34

Repetição 52

**Listas** **70**

Métodos Comuns 71

Quiosque de votações 72

Despesas de projeto 74

**Funções** **76**
Estrutura 76
Exemplos 77
Desafios 78
**Considerações Finais** **81**
**Referências Bibliográficas** **82**
**Lista de figuras** **83**
**Tabelas** **86**

# LINGUAGEM PYTHON E A SUA INTERFACE DE DESENVOLVIMENTO

## Porquê programar em Python?

Na última década, o Python tornou-se uma das linguagens de programação mais populares e amplamente utilizadas. A sua ascensão deve-se, em grande parte, à sua simplicidade e legibilidade, características que fazem da sua sintaxe clara e intuitiva uma grande aliada, especialmente para quem está a dar os primeiros passos na programação. Escrever código em Python é, portanto, um processo acessível e descomplicado. No entanto, à medida que o utilizador aprofunda os seus conhecimentos, descobre que a linguagem também oferece uma notável capacidade para lidar com problemas mais complexos, mostrando-se versátil e poderosa tanto para iniciantes como para programadores mais experientes.

Graças também à sua versatilidade, pode ser usada para uma ampla gama de aplicações de desenvolvimento *standalone*[1], *mobile* e *Web*, análise de dados, inteligência artificial, etc. Possui uma vasta coleção de bibliotecas e *frameworks*[2] para desenvolvimento de projetos mais complexos.

Devido à sua facilidade e à enorme comunidade ativa de utilizadores, é amplamente utilizada em diversos equipamentos para controle e manipulação como, por exemplo, maquinaria industrial.

Por todas estas razões, consideramos o Python a linguagem ideal para iniciar a aprendizagem de programação, especialmente quando acompanhado pelo estudo de algoritmos, criando uma base sólida para o desenvolvimento de competências nesta área.

## IDLE — Integrated and Development Learning Enviroment

Vamos utilizar o **Python 3.0**. A versão 3.0 introduz várias melhorias e novas funcionalidades em relação às versões anteriores.

Para usar uma linguagem de programação de forma simples, é indispensável um IDE (Integrated Development Environment). É uma ferramenta no mundo da programação que reúne num só ambiente todos os recursos necessários para criar software de forma eficiente. Desde a escrita do código até à sua execução e depuração (correção de erros), o IDE facilita o trabalho do programador ao integrar funcionalidades como edição de texto, análise de erros, sugestões automáticas, execução de programas e gestão de projetos. Este tipo de ambiente não só poupa

1 Software que pode ser executado de forma independente, sem a necessidade de estar conectado a outros sistemas ou depender de serviços externos; geralmente é executado no sistema operativo do computador.

2 Aplicação de software com conjunto de ferramentas e bibliotecas utilizadas para o desenvolvimento de software.

tempo, eliminando a necessidade de alternar entre diferentes ferramentas, como também melhora a produtividade e reduz os erros, especialmente em projetos mais complexos.

Iremos usar o **IDLE** (Integrated Development and Learning Environment). É um programa que vem com o Python (download em www.Python.org) e serve como um editor simples para escrever, testar e executar código Python num ambiente simples e intuitivo. É uma ferramenta básica para praticar Python e aprender programação, sem precisar de instalar outras aplicações mais avançadas!

Neste livro, vamos explorar conceitos fundamentais de programação, como variáveis, loops, funções e outros elementos essenciais, utilizando o Python 3.0 no ambiente IDLE. Com esta abordagem, teremos a oportunidade de criar programas que vão desde os mais simples aos mais complexos, desenvolvendo as nossas competências de programação de forma prática, progressiva e, acima de tudo, divertida!

# INTRODUÇÃO À ALGORITMIA

Apesar de parecer recente, a algoritmia (estudo de algoritmos), remonta a milhares de anos atrás. Aliás, a palavra "algoritmo" deriva do nome do matemático persa Al-Khwarizmi (c. 780-850) que escreveu vários livros sobre números e métodos de resolução de equações matemáticas. O seu nome foi traduzido como "Algoritmi", em latim medieval.

No entanto, a ideia de algoritmo é ainda mais antiga, com raízes que remontam às civilizações da Mesopotâmia, Egito e Grécia Antiga, onde eram utilizados métodos repetitivos para resolver problemas matemáticos, como calcular áreas ou resolver equações [1].

Os algoritmos são, essencialmente, conjuntos de instruções organizadas passo a passo, desenvolvidos para realizar tarefas ou resolver problemas de forma sistemática. Embora tenham raízes na matemática, eles são a base da computação moderna, desempenhando um papel central em diversas áreas tecnológicas. Um dos primeiros exemplos de um algoritmo formal é o Algoritmo de Euclides, criado por volta de 300 a.C., utilizado para encontrar o maior divisor comum entre dois números [2].

Com o surgimento dos computadores no século XX, a importância dos algoritmos aumentou de forma exponencial. Alan Turing, considerado um dos fundadores da ciência da computação, introduziu o conceito da "máquina de Turing", na década de 1930. É um modelo teórico que simula o funcionamento de um computador e descreve a execução de algoritmos [3]. Este trabalho lançou as bases para a teoria da computação e a programação moderna.

Na era moderna, os algoritmos estão presentes em praticamente toda a tecnologia. Atualmente, eles são aplicados em diversas áreas, desde as pesquisas realizadas no Google até os sistemas de criptografia usados para garantir a segurança das comunicações online. A linguagem de programação Python, por exemplo, é amplamente utilizada para implementar algoritmos devido à sua simplicidade e versatilidade. Com Python, os programadores de todos os níveis de destreza, conseguem escrever código eficiente e fácil de entender, o que facilita a aplicação de algoritmos em diferentes contextos [4].

## O que é um algoritmo?

Para resolver um problema no computador é necessário que, em primeiro lugar, seja encontrada uma maneira de descrever esse problema de uma forma clara e precisa. É preciso que determinar uma sequência de passos que permitam que o problema possa ser resolvido de maneira automática e repetitiva. Esta sequência de passos é chamada de *algoritmo*.

De modo mais formal, um algoritmo consiste uma sequência finita de instruções bem definidas e não ambíguas, onde cada instrução pode ser executada mecanicamente num período finito e com uma quantidade de esforço limitada. Um algoritmo não representa, necessariamente, um programa de computador, mas sim os passos

necessários para realizar uma tarefa. A sua implementação pode ser feita por um computador, por outro tipo de autómato ou até mesmo por um ser humano.

Algoritmos diferentes podem realizar a mesma tarefa utilizando conjuntos distintos de instruções, podendo variar em termos de tempo, espaço ou esforço necessários. Um exemplo disso são os algoritmos de um GPS: tendo como objetivo ir do ponto A para o ponto B, o GPS apresenta três rotas possíveis: a mais rápida, a mais económica e a que envolve menos quilómetros a percorrer.

A noção de algoritmo é fundamental para toda a computação. A criação de algoritmos para resolver os problemas é uma das maiores dificuldades enfrentadas pelos principiantes em programação de computadores.

## A receita de um bolo ou a representação simples de um algoritmo

A forma mais simples de entender um algoritmo é, provavelmente, compará-lo a uma receita de culinária. Uma receita descreve um conjunto de passos ou ações que combinam vários ingredientes com vista a obter um produto gastronómico particular.

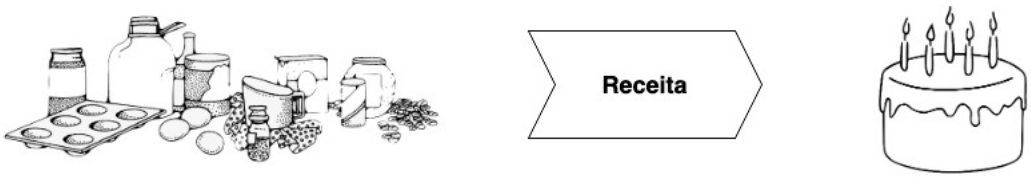



*Figura 1 – Diagrama de transformação de ingredientes culinários para receita de um bolo*

Assim como uma receita opera sobre um conjunto de ingredientes (farinha, ovos, açúcar, etc.) para gerar um produto final (como um bolo), um algoritmo processa um conjunto de entradas para produzir uma saída útil (ou desejada) para o utilizador.

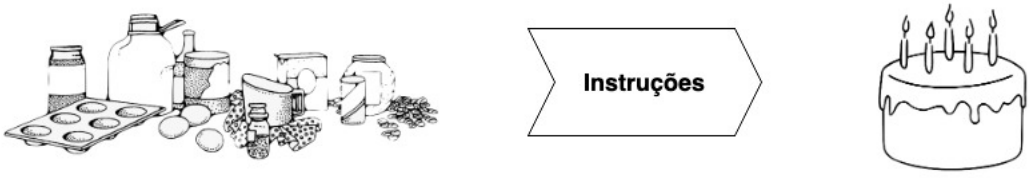



*Figura 2 - Diagrama de transformação de ingredientes culinários com instruções para confeção de um bolo*

Para o caso específico, um algoritmo para fazer um bolo poderia ser descrito da seguinte forma:
1) Bater duas claras em castelo;
2) Adicionar duas gemas;
3) Adicionar uma chávena de açúcar;

4) Adicionar duas colheres de sopa de manteiga;
5) Adicionar uma chávena de sumo de laranja;
6) Adicionar duas chávenas farinha;
7) Adicionar uma colher de café de fermento;
8) Colocar numa forma e levar ao forno a 180° durante 30 minutos.

Neste exemplo, cada passo corresponde a uma instrução do algoritmo, e, tal como num algoritmo de programação, a ordem das instruções e a forma como são executadas é fundamental para o sucesso do processo. Se algum passo for alterado ou executado fora de sequência, o resultado pode não ser o desejado. Assim como um algoritmo computacional processa entradas para produzir uma saída específica, esta receita transforma os ingredientes iniciais no produto final — o bolo pronto para ser saboreado.

## Construção de algoritmos

A construção de algoritmos consiste na definição precisa de uma sequência de passos lógicos para resolver um problema, utilizando uma abordagem específica que permite que um computador ou outro dispositivo execute essas instruções de forma automática e eficiente, gerando a saída desejada.

É importante ter em consideração que a elaboração de programas envolve três fases distintas:

- a análise do problema, que inclui a especificação do problema, a análise de requisitos, pressupostos, entre outros;
- a conceção do algoritmo, que define os passos necessários para resolver o problema;
- a tradução desse algoritmo na linguagem de programação, que permite que o computador execute as instruções de forma eficaz.

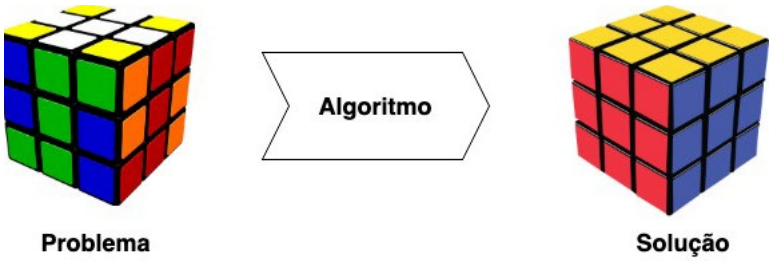



*Figura 3 — O trio Problema, Algoritmo, Solução(Programa)*

É importante sublinhar que a compreensão do problema é a fase mais importante na construção de um algoritmo. Se não entender bem o problema que tem em mãos, poderá conceber um algoritmo incorreto, dando origem a um programa com *bugs* (erros de código). Para mitigar este problema, que é bastante comum e recorrente, apresentamos os passos necessários para a construção de um algoritmo, de forma clara e estruturada (ou seja, sob a forma de algoritmo ☺)

## Passos na construção de algoritmos

- Compreender o problema;
- Identificar os dados de entrada;
- Identificar os dados de saída;
- Determinar o que é necessário para transformar os dados de entrada em dados de saída:
  - usar a estratégia do dividir-para-conquistar;
  - observar regras e limitações;
  - identificar todas as ações a realizar;
  - eliminar ambiguidades;
- Construir o algoritmo;
- Testar o algoritmo;
- Executar o algoritmo.

Uma abordagem eficaz para resolver problemas complexos é a técnica de dividir o problema em partes menores e mais fáceis de tratar: método cartesiano dividir-para-conquistar.

Este método, utilizado na criação de algoritmos [5], é também conhecido como método descendente (*top-down*) ou método de refinamento passo-a-passo. Consiste em dividir um problema em partes menores (subproblemas) para facilitar a sua conceção e resolução. A abordagem passo-a-passo implica que cada passo é completado antes de iniciar o próximo, seguindo uma sequência lógica e dependente. O exemplo anterior da receita de um bolo ilustra bem a aplicação deste método.

## Características fundamentais de um algoritmo

Apesar da bibliografia existente apresentar um conjunto de características mais amplo [6], neste contexto, entendemos que um algoritmo deve possuir as seguintes cinco características fundamentais:

**Finitude**: um algoritmo deve sempre terminar após um número finito de passos.

**Definição**: cada passo de um algoritmo deve ser claramente definido, com ações precisas e sem ambiguidades.

**Entradas:** um algoritmo deve ter zero ou mais entradas, que são dados fornecidos antes do algoritmo iniciar.

**Saídas:** um algoritmo deve produzir uma ou mais saídas, que tem uma relação específica com as entradas fornecidas.

**Eficiência:** um algoritmo deve ser eficiente, ou seja todas as operações devem ser suficientemente básicas de modo que possam ser executadas com precisão e num tempo finito, até mesmo por um ser humano usando papel e lápis.

Ressalva-se que pode haver mais do que um algoritmo para resolver um mesmo problema, como exemplificado pelo caso do GPS, referido anteriormente. A escolha

do algoritmo pode variar dependendo das diferentes abordagens, objetivos ou contextos, que influenciam diretamente a sua conceção e construção.

## Formas de representação de algoritmos

Não existe consenso entre os especialistas sobre qual é a melhor forma de representar um algoritmo. Cada método tem as suas vantagens e é escolhido conforme a necessidade e o contexto do problema a ser resolvido [7].

**Linguagem natural**
Nesta abordagem, os algoritmos são expressos diretamente em linguagem natural, como no exemplo do algoritmo da receita do bolo, apresentado em língua portuguesa.

**Pseudo-linguagem**
Trata-se de uma linguagem intermédia entre a linguagem natural e uma linguagem de programação, usada para descrever os algoritmos de forma mais estruturada. É a abordagem mais comum pois tem a vantagem de permitir que o algoritmo seja escrito de uma maneira próxima à de uma linguagem de programação de computadores.

**Fluxograma (ou Diagrama de fluxo)**
Representação gráfica que utiliza formas geométricas padronizadas para indicar as diversas ações e decisões que devem ser executadas na resolução do problema.

Os fluxogramas são a forma de representação de algoritmo adotada neste livro. Eles facilitam a visualização espacial do problema e são fáceis de elaborar. Além disso, são amplamente utilizados em diferentes contextos, como na indústria e nos serviços.

Como referido, os fluxogramas representam graficamente um algoritmo, descrevendo o fluxo de dados através de um conjunto de figuras geométricas padronizadas conectadas por setas que indicam o fluxo.

As formas geométricas empregues neste livro para a conceção de fluxogramas são as seguintes:

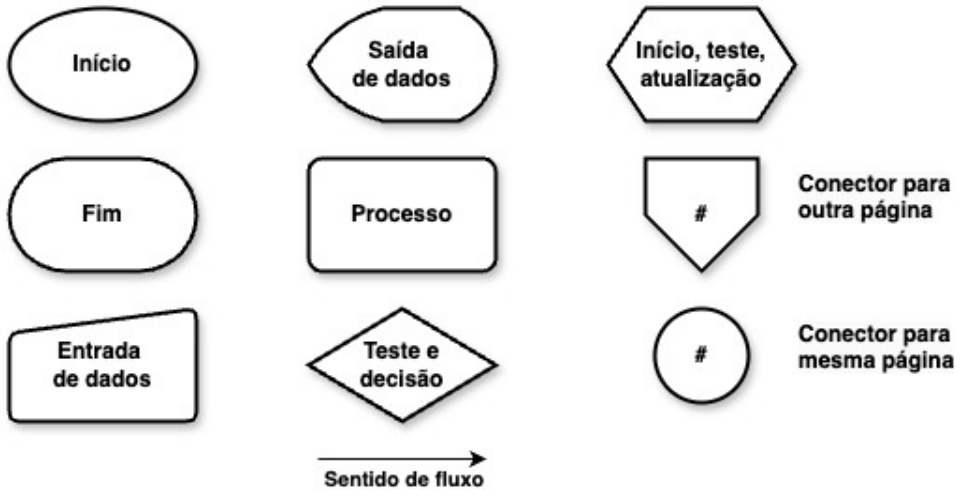



*Figura 4 – Simbologia para a criação de fluxogramas*

É importante realçar que, nesta secção do livro, não abordaremos em detalhe a descrição e a utilização das formas geométricas para a criação de algoritmos. No entanto, essas formas serão utilizadas em contextos apropriados ao longo do livro, proporcionando uma compreensão mais clara e eficiente da sua aplicação.

Ainda, de referir que existem outras formas geométricas para a conceção de algoritmos. No entanto, para os desafios abordados neste livro, as formas selecionadas são perfeitamente adequadas!

Preparado para criar o primeiro algoritmo e programa? Vamos começar!

## O clássico *Hello World!*

O programa "Hello, World!" é, tradicionalmente, o primeiro exemplo que muitos aprendizes de programação escrevem. Tornou-se popular nos primórdios da programação, especialmente com a linguagem C [8].

Trata-se de um programa bastante simples, cujo objetivo é apenas exibir a mensagem "Hello World!" no ecrã do computador. Aqui, vamos fazer exatamente isso: criar um programa com uma única instrução para imprimir a mensagem "Hello, World!". Abaixo apresentamos o fluxograma do algoritmo e o programa em Python.

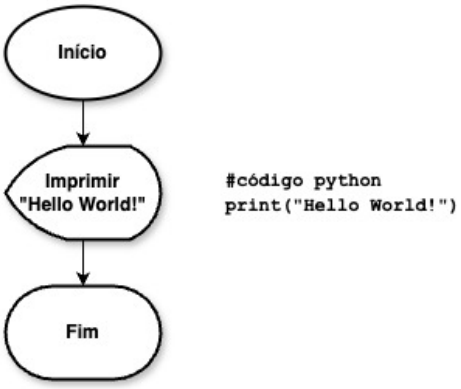



*Figura 5 – Fluxograma do algoritmo e programa "Hello World!"*

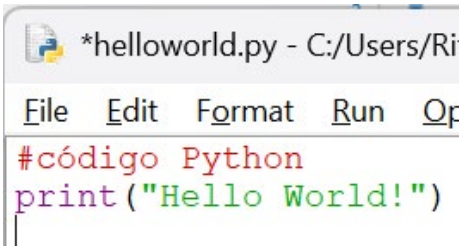
Agora, vamos aprender como escrever e executar o programa "Hello, World!" utilizando o IDLE e a linguagem Python. Caso ainda não tenha o Python e o IDLE instalados no seu computador, será necessário instalá-los antes de prosseguir. Para fazer isso, basta visitar o site oficial https://www.python.org/downloads/ e seguir as instruções de download e instalação.

Ao seguir os passos abaixo, veremos como criar e executar este primeiro programa simples, permitindo uma introdução prática à programação com Python:

Utilizando o IDLE para correr o programa:

1. Abrir o programa IDLE;
2. Abrir um novo ficheiro (**File**->**New File**);

3. Escrever o código apresentado na Figura 5;




4. Gravar o programa (**File**->**Save** ou **File**->**Save As**)
5. Executar o programa (**Run**->**Run Module** ou primir **F5**)

Funcionou?

A instrução `#código Python` não tem interferência na execução do programa; trata-se apenas de um comentário. Em Python, todos os comentários começam pelo símbolo cardinal (#). Pode imaginar um comentário como um *Post-It* ou um apontamento, que serve apenas para adicionar notas ou explicações no código, sem afetar seu funcionamento.

A instrução `print` em Python é utilizada para gerar a impressão de uma mensagem no ecrã. Quando escrevemos `print("Hello, World!")`, o Python exibe a frase "Hello, World!" no terminal ou na janela de execução. Esse comando é responsável por mostrar o que estiver dentro dos parênteses, seja uma frase, variável ou até uma expressão, no ecrã para o utilizador.

Aproveitando o sucesso na execução do seu primeiro algoritmo em Python, vamos agora apresentar alguns conceitos fundamentais que todo programador deve compreender.

# BREVES CONCEITOS DE INTRODUÇÃO À PROGRAMAÇÃO

Um programador deve pensar como *um cientista de computação*. Tal como um matemático, ele utiliza linguagens formais para desenvolver soluções para problemas, mapeando-os em dados e instruções que processam (ou alteram) esses dados. O requisito mais importante para um programador é a capacidade de resolver problemas, ou seja, a habilidade de formular questões, pensar e encontrar soluções claras e assertivas.

Um programador não deve se isolar do mundo real. Pelo contrário, é fundamental que se interesse por tudo o que envolve sua vida profissional. Esse interesse potencia a sua capacidade de aprender sobre diversas áreas e adquirir conhecimentos e habilidades que facilitam a resolução de problemas. Além disso, é essencial que compreenda as linguagens de programação, para que consiga traduzir algoritmos em programas executáveis.

## Categorias de linguagens de programação

As linguagens de programação dividem-se principalmente em duas categorias:

**Linguagens de alto nível**
Estas linguagens são mais próximas da linguagem humana, com sintaxe mais fácil de ler e escrever. São exemplos as linguagens Python, C++, Java e PHP.

**Linguagens de baixo nível**
Estas linguagens são orientadas para a comunicação direta com o hardware do computador. São mais complexas pois exigem um entendimento profundo da arquitetura da máquina. Exemplos comuns destas linguagens são Assembly, Forth e código de máquina.

Nos primórdios da computação, na década de 1950, as linguagens de programação predominantes eram as de baixo nível. No entanto, apresentavam alta complexidade de compreensão e eram extremamente dependentes da arquitetura específica de cada computador. Isso significava que um programa escrito para uma máquina dificilmente funcionaria noutra. Com a evolução da computação, surgiram as linguagens de alto nível, tornando o processo mais acessível e produtivo.

Contudo, os computadores só entendem a linguagem de máquina (código binário). Portanto, as linguagens de alto nível necessitam passar por um processo de tradução para que o hardware possa executar as instruções. Essa conversão é realizada por dois tipos principais de programas: interpretadores e compiladores.

Um **interpretador** lê e executa o código (*source code*) diretamente, sem gerar um arquivo separado. Ele processa o programa gradualmente, traduzindo e executando cada linha do código imediatamente. Além disso, os erros são geralmente detetados

à medida que o código é interpretado, o que pode tornar o processo de *debugging*[3] mais simples e eficiente.

*Figura 6 – Execução de um programa por um interpretador [9]*

Um **compilador** lê o código-fonte (*source code*), converte-o em código de máquina (*object code*) e, posteriormente, executa-o. Ele traduz na íntegra o programa antes que este seja executado. Durante o processo de compilação, o compilador verifica todo o código em busca de erros e aplica otimizações. Após a compilação, é gerado um arquivo executável, que pode ser executado várias vezes sem necessidade de recompilação, excepto se o código (*source code*) for alterado.

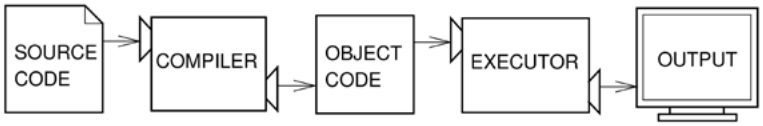



*Figura 7 - Execução de um programa por um compilador [9]*

O Python é uma linguagem de programação interpretada, podendo o seu código ser executado de duas formas: *modo interativo*, onde os comandos são executados imediatamente, e *modo de script*, em que o código é salvo em arquivos e executado como um todo (fazendo uso do IDLE).

## Termos comuns na programação

Um programa é uma sequência de instruções que define como executar um processo computacional. Esse processo pode envolver cálculos matemáticos, como resolver equações ou simplificar polinómios, mas também possa executar computação simbólica, como pesquisar texto, aceder a bases de dados, entre outros.

Independentemente da linguagem de programação utilizada para desenvolver o programa, existem certas instruções fundamentais comuns a todas:

**Input (Entrada de dados):** obter dados de fontes externas como um teclado, ficheiro ou sensores; por exemplo, o programa pede ao utilizador para introduzir o seu nome.

**Output (Saída de dados):** enviar informações para um monitor, ficheiro ou outro dispositivo; por exemplo, o programa exibe "Bem-vindo, Miguel" no ecrã.

**Math (Operações matemáticas)**: realizar operações aritméticas básicas, como

3 Processo de análise e tratamento de erros

somar números, calcular médias; por exemplo, um programa que calcula o salário líquido com base no salário bruto.

**Execução condicional:** avaliar condições específicas para decidir quais instruções executar; se algo for verdadeiro, executa uma ação, caso contrário, faz outra coisa; por exemplo, "Se o utilizador tiver mais de 18 anos, permitir acesso; caso contrário, mostrar uma mensagem de erro."

**Repetição (Loops):** repetir uma série de ações, de forma automática, seja por um número definido ou enquanto uma condição for verdadeira; por exemplo, um programa que pergunta o nome de cada participante de uma lista até terminar.

## *Debugging* – Deteção de erros

Programar resulta quase sempre em erros. Esses erros são chamados de **bugs** e o processo de os identificar é conhecido como ***debugging***.

O termo *bug* tem uma origem interessante. A história remonta a 1947, quando uma equipa de cientistas da computação, liderada por Grace Hopper, da Universidade de Harvard, estava a trabalhar no computador *Mark II*. Durante os testes, encontraram um erro que impedia o computador de funcionar corretamente. Ao investigar, descobriram que uma traça tinha ficado presa num dos componentes do computador, causando o problema. A partir daí, a equipa registou o incidente como "*first actual case of bug being found*" (primeiro caso real de um "bug" encontrado) [9]. Desde então, o termo *debugging* passou a designar o processo de encontrar e corrigir erros nos programas de computador.

Existem vários tipos de erros de programação. Iremos apenas ver os três principais.

**Erros de sintaxe (Syntax errors)**
Este tipo de erros ocorrem quando o código escrito não segue as regras gramaticais da linguagem de programação utilizada. Por exemplo, esquecer de fechar os parenteses**:**

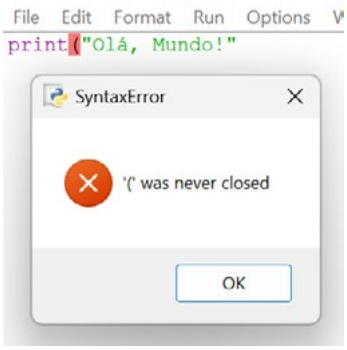



*Figura 8 – Erro de sintaxe no IDLE (parêntesis em falta)*

Para evitar este tipo de erros devemos ler e entender a sintaxe da linguagem que estamos a utilizar, ler as mensagens de erro fornecidas pelos compiladores ou interpretadores pois estas geralmente apontam exatamente onde o erro ocorreu, o que pode ajudar a corrigir o problema rapidamente.

**Erros de execução (Runtime errors)**
Este tipo de erros ocorrem apenas quando o programa está a ser executado, ou seja, depois de o código ser compilado ou interpretado. Esses erros acontecem por várias razões, e, normalmente, o programa ainda corre até o ponto onde ocorre o erro, mas, ao tentar realizar uma operação inválida ou inesperada, ele falha e gera uma mensagem de erro.

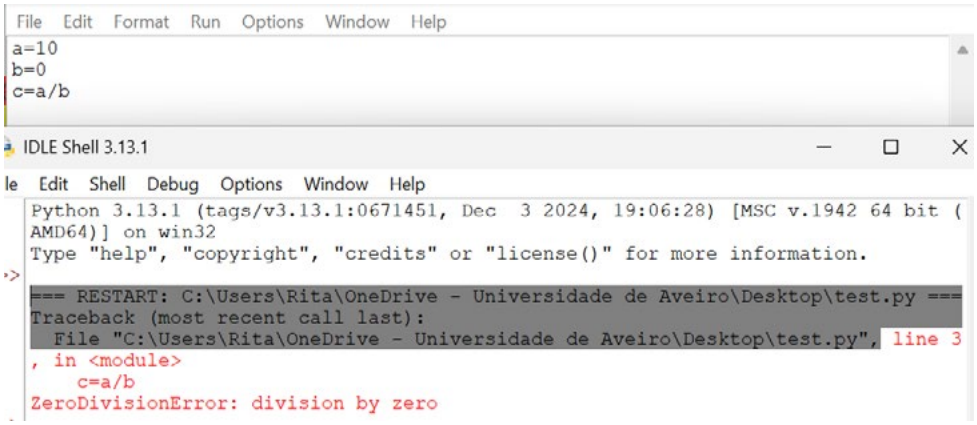



*Figura 9 — Erro de execução no IDLE (divisão impossível por zero)*

Tal como nos erros de sintaxe, devemos estar atentos à mensagem de erro que indica o tipo de problema e analisar essas mensagens para perceber o que causou o erro e onde ocorreu no código.

**Erros semânticos (Semantic errors):**
Também conhecidos como erros de lógica, ocorrem quando o programa é executado corretamente (ou seja, sem erros de sintaxe ou de execução), mas não faz o que era esperado. Estes erros surgem porque o programa foi mal construído e, apesar de estar a funcionar, não faz aquilo que se pretendia.

Erros semânticos são complicados porque não causam falhas diretas no programa, mas sim resultados errados. Corrigi-los exige um bom entendimento dos objetivos do programa e testes cuidadosos para garantir que o comportamento do código é o esperado.

## Variáveis

No mundo da programação, as **variáveis** são um dos conceitos fundamentais e servem como base de qualquer programa. Elas permitem que os programadores armazenem informações temporariamente na memória do computador e as utilizem ao longo da execução do código. Sem variáveis, seria impossível criar programas dinâmicos, uma vez que não haveria um jeito de guardar dados como números, textos ou até mesmo resultados de cálculos para serem usados posteriormente.

Independentemente da linguagem de programação que estás a usar, o conceito de variável é sempre o mesmo: uma "gaveta" para armazenar dados. No entanto, cada linguagem tem as suas próprias regras para trabalhar com elas.

Uma variável é um nome simbólico que é utilizado para armazenar dados que podem ser manipulados ou referenciados ao longo do programa. Pensa nela como uma

"gaveta" etiquetada onde guardas informações para usar mais tarde. O conteúdo dessa "gaveta" pode mudar durante a execução do programa (daí o nome variável).

As componentes de uma variável incluem o nome, que é o rótulo atribuído à variável, como, por exemplo, "idade", "nome" ou "altura". Este nome deve seguir regras específicas da linguagem de programação, como não começar com números em muitas linguagens. Outra componente é o valor, ou seja, o dado que está armazenado na variável, como "25", "João" ou "1.75".

Para estabelecer a relação entre o nome da variável e o seu valor numa linguagem de programação usamos normalmente o operador =:

```
nomeDaVariavel = valorDaVariavel
```

Por exemplo, se queremos criar uma "gaveta" etiquetada com o nome "idade" e que guarda o valor 20, criamos no programa com o comando "`idade=20`", como mostra a Figura 8, e não "`20=idade`".

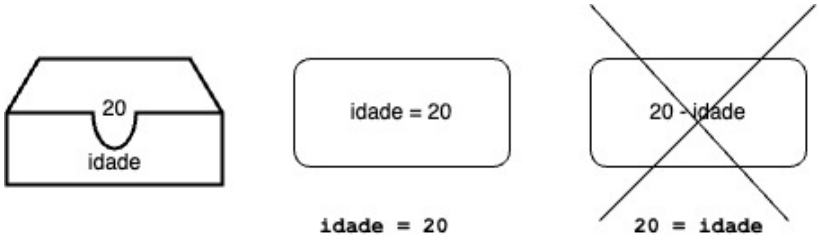



*Figura 10 – Exemplo de instrução de criação de uma variável*

**Tipos de variáveis**

A última componente associada a uma variável é o tipo da variável. O tipo de dados define o tipo de informação que uma variável pode armazenar. Por exemplo, variáveis do tipo **inteiro** (*int*), guardam números inteiros, enquanto as do tipo **decimais** (*float* ou *double*) armazenam números decimais. As variáveis do tipo **string** (*str*) armazenam sequências de texto. Dependendo da linguagem de programação, o tipo pode ser explícito, sendo declarado pelo programador, ou implícito, sendo inferido automaticamente pela linguagem. Por exemplo, na linguagem Python, basta atribuir um valor à variável e o tipo é inferido automaticamente.

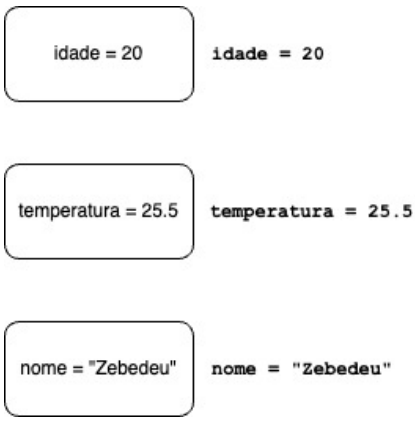



*Figura 11 – Exemplos de tipos de variáveis*

Existem outros tipos de variáveis, tais como *booleanas* (true/false), character (*char*) e outros, mas os três tipos de variáveis inteiro, decimal e texto servirão para a maioria dos assuntos e desafios apresentados ao longo do livro.

**Nomenclaturas de variáveis**

As regras para atribuir um nome a uma variável podem variar ligeiramente dependendo da linguagem de programação, mas, de forma geral, existem algumas diretrizes comuns que devem ser seguidas.

Em primeiro lugar, o nome da variável deve começar com uma letra ou um *underscore* (_), não sendo permitido iniciar com números. Além disso, os nomes podem conter letras, números e sublinhados, mas não podem incluir espaços, cedilhas, acentos ou caracteres especiais (como @, #, % ou -). Outra regra importante é que as palavras da própria linguagem de programação, como por exemplo *if*, *while* ou *class*, não podem ser usadas como nomes de variáveis.

Na maioria das linguagens, os nomes de variáveis são sensíveis a maiúsculas e minúsculas (*case-sensitive*), o que significa que "idade" e "Idade" seriam considerados nomes diferentes.

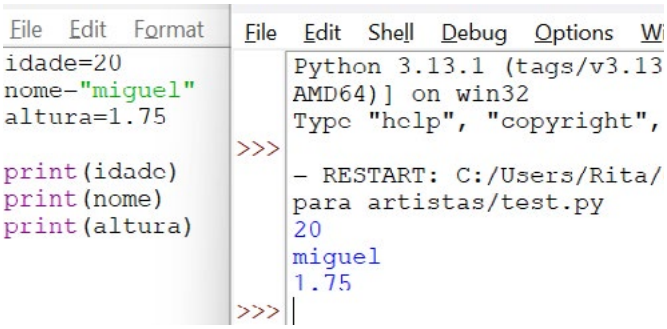



*Figura 12 – Criação de variáveis no IDLE*

É importante usar boas práticas na nomeação das variáveis para que o código organizado, claro e fácil de compreender. Por exemplo, escolher nomes para as variáveis que reflitam o propósito da variável, evitar nomes excessivamente curtos ou abreviaturas confusas, entre outras.

Existem diferentes estilos de nomenclatura que podem ser seguidos dependendo da linguagem ou do contexto:

**snake_case**: os nomes são palavras com todas as letras minúsculas e as palavras separadas por *underscores*; por exemplo, os nomes `nome_do_utilizador` e `idade_do_utilizador`.

**camelCase**: os nomes são as palavras que são unidas e cada palavra, exceto a primeira, começa com uma maiúscula; por exemplo, os nomes `nomeDoUtilizador` e `idadeDoUtilizador`.

**PascalCase**: *apenas* difere do *camelCase* no sentido de iniciar o nome com maiúscula; por exemplo, `NomeDoUtilizador` e `IdadeDoUtilizador`.

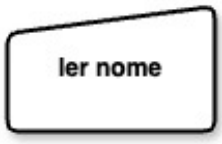
Para os algoritmos e programas que vamos apresentar e desafiar o leitor, a nomenclatura que adotamos é a *camelCase*. Existem vários fundamentos sobre qual a

nomenclatura a utilizar [10], no entanto, a nossa opção deve-se a vários fatores, nomeadamente por esta ser uma nomenclatura universal e das mais utilizadas, permitir uma melhor contextualização dos dados a guardar e uma melhor legibilidade da variável.

## *Input* (entrada de dados)

O *input* é qualquer informação que o programa obtém de um utilizador, de um ficheiro, de uma base de dados ou de outra fonte externa, como o rato ou o microfone. Estas entradas permitem que o programa processe dados diferentes em cada execução, em vez de trabalhar apenas com valores fixos.

Neste livro, salvo alguma exceção, um *input* será feito através de um teclado. O seu símbolo e instrução em fluxograma, serão do tipo:




```
nome = input("Indique o seu nome")
```

*Figura 13- Instrução de input*

Aqui podemos observar algumas diferenças entre o objeto do fluxograma e o código em Python. No fluxograma, o símbolo indica que será realizado um *input* por teclado, e que essa entrada ficará guardada numa variável denominada `nome`. No Python, indicamos que a variável `nome` será igual ao dado introduzido pelo utilizador através da instrução `input()`. Esta instrução apresenta a mensagem que está entre aspas, para contextualizar e informar o utilizador no sobre o que se pretende. A informação inserida pelo utilizador será então armazenada na variável `nome`.

## Output (saída de dados)

No exemplo do "O clássico Hello World!", vimos um exemplo de saída de um output. O *output* é a forma como o programa apresenta os seus resultados ou informações ao utilizador ou a outros sistemas. Ele pode ser exibido no ecrã, gravado num ficheiro, apresentado graficamente ou enviado para uma impressora.

Neste livro, salvo alguma exceção, o *output* será direcionado para um ecrã. O seu símbolo e instrução em fluxograma, serão:

```
print("Olá")
```

*Figura 14- Instrução de output*

## *Inputs* e *Outputs* (exemplos)

Vamos analisar dois exemplos simples que combinam *inputs* e *outputs*.

O objetivo será criar um algoritmo e um programa que solicite ao utilizador o nome e idade. Posteriormente, o programa irá apresentar uma mensagem a cumprimentar o utilizador e dizendo-lhe a sua idade. O resultado no ecrã (output) seria algo semelhante a:

```
Por favor, diga-me o seu nome: Miguel
Diga-me a sua idade: 50
Olá Miguel. Tem 50 anos.
```

*Figura 15 - Execução do programa com input e output*

O fluxograma para o algoritmo e o código em Python seriam:

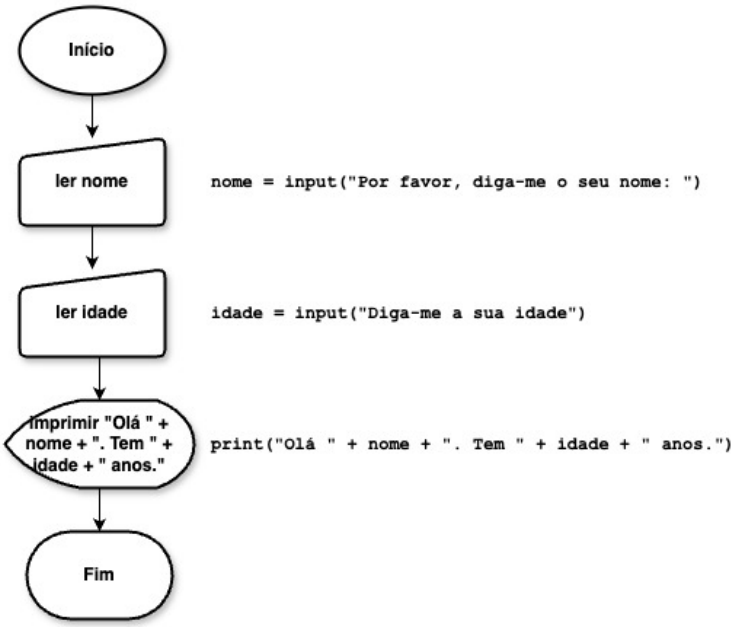



*Figura 16 – Fluxograma do algoritmo e o programa do nome e idade*

Reparou que uma instrução de impressão, tanto em algoritmo como em Python, pode ser composta por vários dados? Esses dados podem ser estáticos, como palavras ou números fixos (no exemplo, `"Olá"`, `"Tem"`, `"anos"`), e também podem ser dinâmicos, como as variáveis (no exemplo, `nome` ou `idade`). Para imprimir uma mensagem que combine esses dois tipos de dados, podes usar o caracter `+`, que serve para juntar esses elementos. Em Python, escreveria:

```
File  Edit  Format  Run  Options  Window  Help
nome=input("Por favor, diga-me o seu nome: ")
idade=input("Diga-me a sua idade: ")

print("Olá " + nome + ". Tem " + idade + " anos.")
```

*Figura 17 - Programa do nome e idade no IDLE*

Este processo de juntar texto e variáveis permite criar mensagens dinâmicas que podem ser personalizadas conforme os valores das variáveis.

### Cálculo de preço de artigo com IVA

Vejamos mais um exemplo. Pretendemos calcular o valor de um artigo com taxa de IVA. O valor sem IVA e a taxa do IVA são inseridos pelo utilizador, e programa apresenta o valor já com o IVA incluído. O resultado seria semelhante a:

```
Introduza o valor do artigo: 25
Introduza a taxa do IVA: 6
O artigo de 25.0 euros com 6.0% de IVA custa 26.5 euros.
```

*Figura 18 - Execução do programa do cálculo do IVA sob um artigo*

Vejamos agora o fluxograma do algoritmo e o programa em Python.

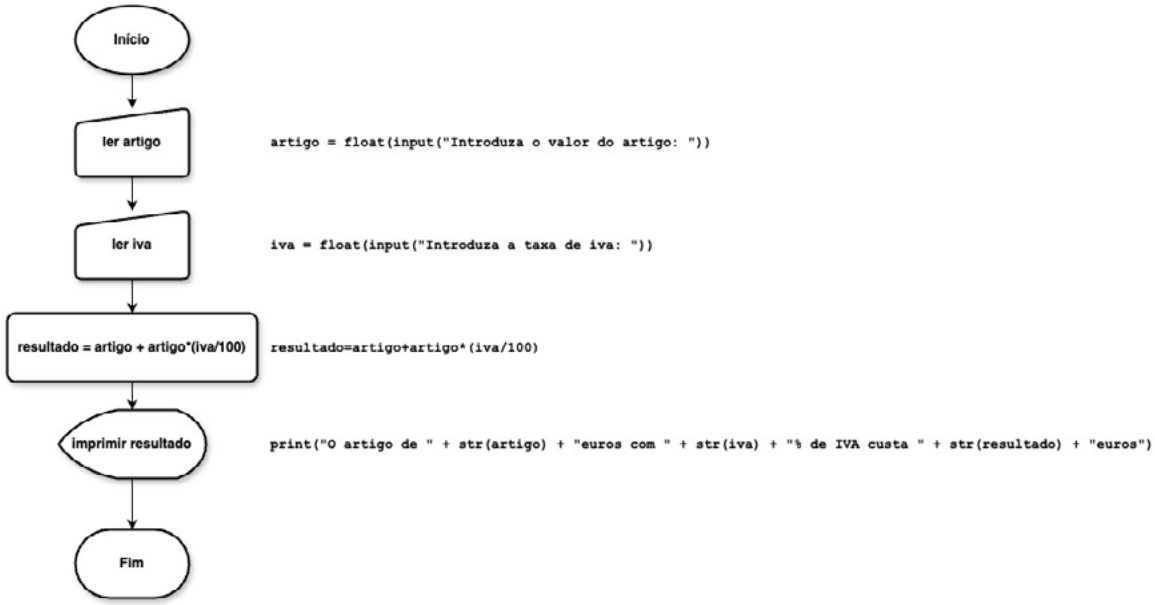



*Figura 19 — Fluxograma e programa em Python para o cálculo de um preço com taxa de IVA*

Gostaríamos de destacar alguns pontos importantes neste algoritmo e programa. O primeiro ponto refere-se à simplificação do fluxograma. Na instrução de *output* (`imprimir resultado`), optámos por simplificar o texto comparativamente à instrução em Python, de modo a tornar o fluxograma mais legível. O principal objetivo do fluxograma é mostrar o processo de cálculo, enquanto que, no Python, criámos uma instrução mais completa para garantir contexto e clareza para o utilizador, concatenando as variáveis e os dados estáticos.

Deve ter reparado que os dois inputs (valores de `artigo` e `iva`) são convertidos em tipo decimal, através do comando `float()`. Isto acontece porque a função `input()` regista sempre os dados em tipo `string` (sequências de caracteres). Ou seja, quando precisamos de realizar cálculos com números decimais, temos de converter esses *inputs* para tipo decimal através da função `float()` ou não conseguiremos realizar as nossas operações. Caso contrário, o Python não conseguiria realizar as operações matemáticas corretamente.

Ainda na instrução de *output*, como as variáveis `artigo`, `iva` e `resultado` são do tipo decimal, temos de as converter para tipo `string` utilizando a função `str()`. Isto é necessário porque a função `print()` quando congrega diferentes tipos de dados, numéricos e texto, apenas os consegue imprimir em tipo `string`. Neste caso, para imprimir e concatenar as variáveis `artigo`, `iva` e `resultado` com o texto fixo, convertemos esses os valores para `string`.

Note-se ainda que na instrução que calcula a variável `resultado`, utilizamos o asterisco (`*`). O asterisco é o símbolo utilizado em programação para representar a multiplicação. Este é o formato padrão para a operação de multiplicação em Python e ajuda a evitar confusão com o caracter "X".

## Operadores em programação

Os operadores são uma parte fundamental de qualquer linguagem de programação. Estes permitem realizar diversas operações sobre dados, desde cálculos matemáticos básicos até comparações e atualizações de valores em variáveis. Podem ser classificados em diferentes categorias consoante sua função.

**Operadores aritméticos** são utilizados para realizar operações matemáticas, como soma, subtração, multiplicação e divisão.

**Operadores de atribuição** permitem guardar ou atualizar valores em variáveis de forma eficiente.

**Operadores de comparação** são usados para avaliar condições, verificando, por exemplo, se dois valores são iguais ou se um é maior que o outro.

**Operadores lógicos** são usados para combinar ou modificar expressões condicionais, permitindo a criação de testes lógicos mais complexos.

Dominar o uso destes operadores é fundamental para implementar lógica mais avançada e tornar o código mais dinâmico e funcional. Nesta secção iremos explorar cada um desses operadores, ilustrando o seu funcionamento, com exemplos práticos.

## Operadores aritméticos em programação

Os operadores aritméticos em programação, nomeadamente em Python, permitem realizar cálculos matemáticos tais como soma, subtração, multiplicação, divisão, potenciação, divisão inteira e resto da divisão.

Estes operadores trabalham com inputs do tipo inteiro e decimal.

| Operação matemática | Operador em Python |
|---|---|
| **Adição** | + |
| **Subtração** | - |
| **Multiplicação** | * |
| **Divisão** | / |
| **Divisão inteira** | // |
| **Potenciação** | ** |
| **Resto de divisão** | % |

*Tabela 1 - Operadores aritméticos*

Vejamos agora alguns exemplos de utilização em Python.

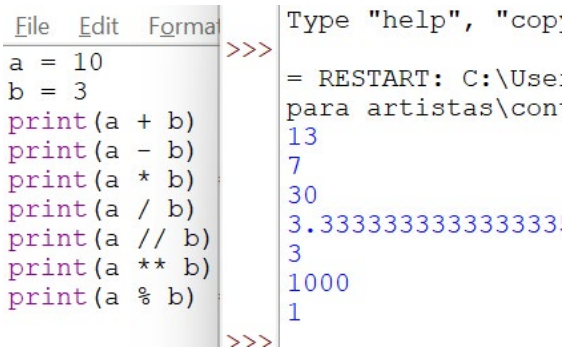



*Figura 20 – Exemplo de código Python com operadores aritméticos*

### O operador aritmético de resto da divisão `%`

A operação resto da divisão ser usado como operador em programação existe por uma razão bastante simples e prática: permite analisar o resto de uma divisão entre dois números. Este operador é muito usado em matemática e em programação, especialmente em situações que envolvem padrões e divisibilidade. Um dos exemplos mais comuns é determinar se um número é ímpar ou par. Ao calcular o resto da divisão de um número `x` por `2`, é possível saber se ele é ímpar ou par através do resto: se o resto da divisão de `x` por `2` for `0`, o número é par; caso contrário, o número é ímpar.

## Operadores de atribuição em programação

Os operadores de atribuição simplificam a escrita de código em operações aritméticas. Tal como o nome diz, atribuem um determinado valor para uma variável, ou seja, são usados para armazenar valores em variáveis. O operador mais básico, e que já usámos anteriormente, é o '=', que atribui diretamente um valor, (recorde que `nomeDaVariável = valorDaVariável`) mas existem outros que combinam a atribuição com as operações matemáticas. Alguns operadores de atribuição estão na tabela abaixo.

| Tipo de atribuição | Código Python | Operação |
|---|---|---|
| Simples[4] | `x = y` | `x = y` |
| Incremento[4] | `x += y` | `x = x + y` |
| Decremento[4] | `x -= y` | `x = x - y` |
| Multiplicação | `x *= y` | `x = x * y` |
| Divisão | `x /= y` | `x = x / y` |
| Divisão inteira | `x //= y` | `x = x // y` |
| Potência | `x **= y` | `x = x ** y` |
| Resto | `x %= y` | `x = x % y` |

*Tabela 2 - Operadores de atribuição*

Um exemplo de um programa em Python destes operadores está na figura seguinte.

File Edit Format Run Options Window Help

```
x = 5  # Atribuição simples
x += 3 # Equivalente a: x = x + 3
print(x)  # Resultado: 8

x *= 2 # Equivalente a: x = x * 2
print(x)  # Resultado: 16

x -= 4 # Equivalente a: x = x - 4
print(x)  # Resultado: 12

x /= 6 # Equivalente a: x = x / 6
print(x)  # Resultado: 2

x **= 7 # Equivalente a: x = x ^ 7
print(x)  # Resultado: 128

x //= 15 # Equivalente à divisão interira de x por 15
print(x)  # Resultado: 8

x %= 3 # Equivalente ao resto a divisão inteira de x por 3
print(x)  # Resultado: 2
```

*Figura 21 – Exemplo de código Python com operadores de atribuição*

Existem outros operadores de atribuição embora, não sejam necessários para o trabalho que estamos a fazer. Eventualmente, no futuro e com maior perícia e especialização, poderão ser úteis para a resolução de diversos problemas.

## Operadores de comparação em programação

Um operador de comparação é usado para comparar valores e variáveis, devolvendo um valor booleano (verdadeiro (`True`) ou falso (`False`)), consoante a compa-

4 Apesar de existir e ser possível adotar todas estas opções na atribuição de valores, estes são os mais comuns a empregar.

ração que se está a efetuar. Eles são muito úteis em estruturas condicionais, como veremos mais á frente.

Alguns de operadores de atribuição estão na tabela abaixo.

| Tipo de Comparação | Símbolo na matemática | Código Python |
|---|---|---|
| Igual | = | == |
| Maior que | > | > |
| Maior ou igual que | □ | >= |
| Menor que | < | < |
| Menor ou igual que | □ | <= |
| Diferente de | □ | != |

*Tabela 3 - Operadores de comparação*

Vejamos um exemplo de aplicação dos operadores de comparação, utilizando instruções concretas com dados.

```
File  Edit  Format  Run  Options  Window  Help
x = 10
y = 5

print(x == y)   # False, porque 10 não é igual a 5
print(x > y)    # True, porque 10 é maior que 5
print(x >= y)    # True, porque 10 é maior ou igual que 5
print(x != y)   # True, porque 10 é diferente de 5
print(x < y)   # False, porque 10 não é menor que 5
print(x <= y)   # False, porque 10 não é menor ou igual que 5
```

```
>>>
= RESTAR
para art
False
True
True
True
False
False
```

*Figura 22 – Exemplo em código Python de operadores de comparação*

### Operador de atribuição = versus operador de comparação ==

Na programação, a forma que se encontrou para distinguir uma *atribuição de valor* de uma *comparação de valor* foi feita do seguinte modo: Enquanto uma atribuição de valor utilizamos um caracter igual (=), na comparação utilizamos dois caracter igual (==), sendo que esta distinção em sintaxe é válida para quase todas as linguagens de programação. Mais à frente veremos alguns exemplos.

A partir deste momento, dispomos das ferramentas necessárias para avançar na programação. Para isso, iremos abordar, de forma genérica, as principais estruturas de controlo, utilizadas na conceção de algoritmos e programas. Posteriormente analisaremos cada uma dessas estruturas de forma individual, acompanhadas de desafios específicos para a sua aplicação.

## Operadores lógicos em programação

Um operador lógico é usado para comparar ou combinar valores e variáveis, devolvendo um valor booleano (verdadeiro (`True`) ou falso (`False`)), consoante a comparação que se está a efetuar [11]. Eles são muito úteis em estruturas condicionais, como veremos mais á frente.

| Operador | Descrição | Exemplo |
|---|---|---|
| `and` | Devolve `True` se ambas as expressões forem verdadeiras | `x<5 and y>=3` |
| `or` | Devolve `True` se uma das expressões forem verdadeiras | `x<5 or y>=3` |
| `not` | Inverte o valor lógico da expressão, transformando `True` em `False` e vice-versa | `not(x<5 or y>=3)` |

*Tabela 4 - Operadores lógicos*

# ESTRUTURAS DE CONTROLO EM PROGRAMAÇÃO

Estruturas de controlo são blocos fundamentais em programação que permitem controlar o fluxo de execução de um programa, tornando-o mais dinâmico e adaptável a diferentes condições ou situações. Elas permitem ao programa tomar decisões, executar blocos de código repetidamente ou interromper a execução com base em determinados critérios [11]. Em meados da década de 1960, matemáticos demonstraram que qualquer programa pode ser construído através da combinação de três estruturas básicas: *sequência*, *seleção* e *repetição* [14].

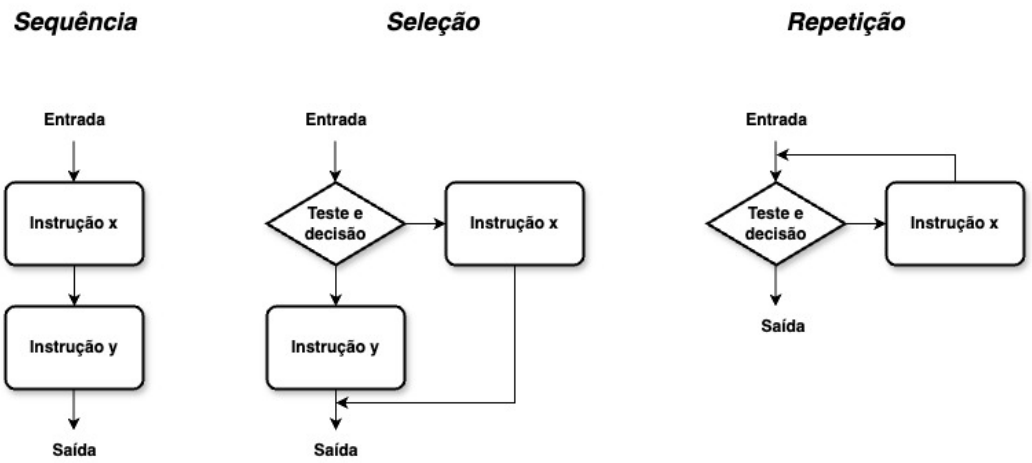



*Figura 23 – Estruturas de controlo em programação*

As estruturas sequência, seleção e repetição são fundamentais na criação de algoritmos e são traduzidas para instruções de programação, como iremos ver. Cada uma destas estruturas pode ser utilizada de forma simples e isolada ou combinada e complementada com as outras, consoante a complexidade e as necessidades do problema em questão. De seguida, exploraremos cada uma destas estruturas de forma individual, para que o leitor possa compreender o seu funcionamento e fluxo de dados que delas resulta.

## Sequência

A estrutura de controlo sequência é a mais simples e fundamental das estruturas de controlo em programação.

Como o nome indica, numa sequência é processado um conjunto de ações (ou instruções) sequencialmente, ou seja, consiste na execução linear e ordenada de instruções, uma após a outra, tal como aparecem no código, sem desvios ou repetições. Não há qualquer possibilidade de alterar a ordem de processamento das ações: após processar a primeira ação, processa-se a segunda, depois da segunda processa-se a terceira, e assim sequencialmente por diante até processar a última ação.

*Figura 24 – Estrutura de controlo Sequência*

Embora as estruturas de controlo estejam a ser introduzidas apenas agora, na secção "Inputs e Outputs (exemplos)", o leitor já se deparou com dois exemplos de algoritmos e programas em que a estrutura de controlo sequência está presente. Ao rever esses exemplos, é importante ter em mente que a ordem das instruções não pode ser alterada. Caso contrário, o programa deixaria de realizar corretamente as operações pretendidas, resultando não só em resultados incorretos, mas também, possivelmente, em erros semânticos ou falhas no sistema.

Vejamos um exemplo de aplicação de uma sequência com Python.

```
File  Edit  Format  Run  Options  Window  Help
# Sequência simples
print("Bem-vind@!")
nome = input("Qual é o teu nome? ")
print("Olá, " + nome + "! É um prazer conhecer-te!")
```

*Figura 25 – Exemplo com código Python de uma estrutura sequência*

Neste exemplo, cada linha é executada na ordem em que aparece:

1. mostra a mensagem de boas-vindas.
2. lê o nome do utilizador.
3. exibe uma mensagem personalizada com o nome fornecido.

De notar que, a sequência anterior de três instruções não pode ser alterada. Não faz sentido em imprimir uma mensagem personalizada com o nome do utilizador se antes não tiver pedido atempadamente essa dado a guardar na variável `nome`. E cada instrução é executada apenas quando a anterior termina.

Embora seja simples, a sequência raramente aparece isolada em programas mais avançados, pois é geralmente combinada com as outras estruturas de controlo para tornar o código dinâmico e adaptável. Mesmo assim, é a base para compreender as outras estruturas.

# Seleção

Também conhecida por *teste e decisão* ou *estrutura de decisão*, a seleção permite que o fluxo do programa continue apenas pela via que é validada por um determinado teste lógico (`True` ou `False`). A seleção pode ter uma, duas ou mais vias sendo que, o fluxo de dados apenas continua única e exclusivamente por uma das vias. Vejamos as três subestruturas:

## Seleção de uma via

Na seleção de uma via, só uma ação dentro desta estrutura é processada, ou seja, existe apenas uma via que pode ser (ou não) executada.

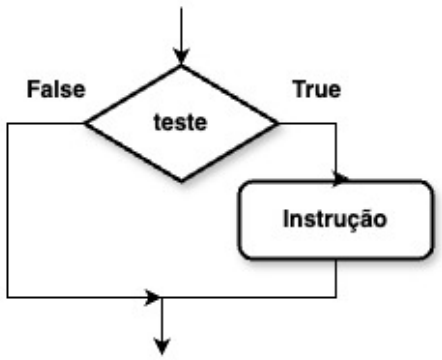



*Figura 26 – Estrutura de seleção de uma via*

Dependendo do teste lógico realizado, o fluxo de dados poderá estar sujeito à execução de instruções adicionais ou, continuar o seu fluxo normal, sem passar por essas instruções.

## Cálculo do preço do menu

Vejamos um exemplo simples: Imagine que um quiosque de um restaurante apresenta o preço de 5€ de um menu de refeição. Se o cliente for maior ou igual a 16 anos, o menu deverá ter um valor acrescido de 2€ do que o cliente com menos de 16 anos, ou seja 7€. O fluxograma e programa seriam:

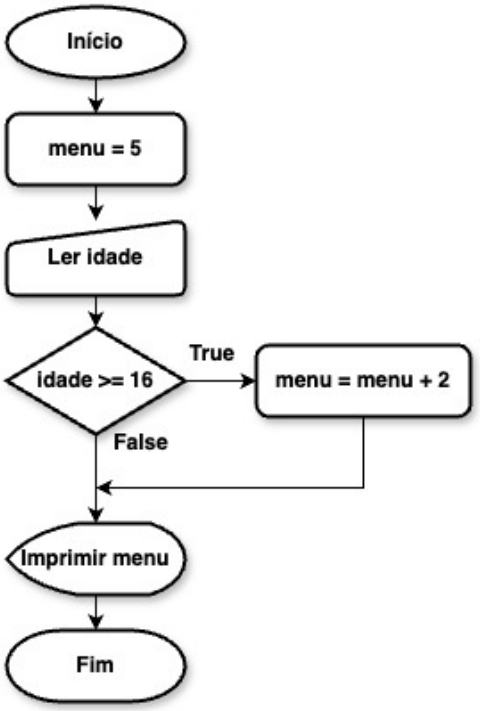



*Figura 27 - Algoritmo de apresentação de preço de menu*

Observe que o valor guardado na variável `menu`, apenas é alterado (são somadas 2 unidades) **se** (`if`) a idade inserida pelo utilizador é maior ou igual a 16. Caso contrário, a variável menu mantém-se com 5 unidades (que representam os 5 €). Traduzindo o algoritmo para Python teríamos algo semelhante ao programa seguinte:

```
menu=5
idade=int(input("Indica a tua idade: "))
if idade>=16:
    menu=menu+2
print("Preço do menu = "+str(menu)+"€")
```

```
== RESTART: /Users/miguel/Desktop/Work/ESAN/Ensino/__boo
k/selecaoIvia_menu.py ==
Indica a tua idade: 17
Preço do menu = 7€
>>>
== RESTART: /Users/miguel/Desktop/Work/ESAN/Ensino/__boo
k/selecaoIvia_menu.py ==
Indica a tua idade: 13
Preço do menu = 5€
>>>
```

*Figura 28 – Programa Python do preço do menu e execução e execução*

**Indentações em Python**

Uma identação é o recuo ou avanço de linhas de texto de um documento, que é utilizado para organizar e estruturar o conteúdo de forma clara. Em programação, a indenação é especialmente importante, pois ajuda a definir blocos de código, tornando-os mais legíveis e compreensíveis.

No exemplo de cima, a instrução de incrementação da variável menu (`menu=menu+2`) está "indenada" no `if`, que representa a estrutura de seleção de uma via. Isto acontece porque o incremento à variável `menu` só acontecerá se for verdade o teste lógico (`if idade>=16:`). Esta é a razão da indenação. Em linguagens como Python, a indenação correta é essencial para a correta execução do código, já que ela indica quais linhas pertencem a uma estrutura de seleção ou repetição. Sem a indenação, o programa resultará num *bug* de sintaxte:

```
menu=5
idade=int(input("Indica a tua idade: "))
if idade>=16:
menu=menu+2
print("Preço do menu = "+str(menu)+"€")
```

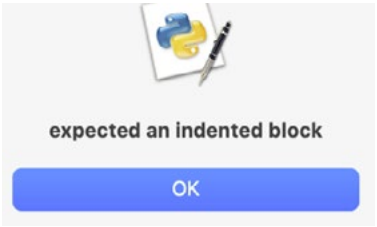



*Figura 29 - Erro de falta de indentação não expectável*

### Seleção de duas vias

A seleção de duas vias difere da seleção de uma via, no sentido de abrir sempre dois caminhos de instruções distintos que, independentemente do resultado do teste lógico, garante sempre que o programa flui por uma das vias com instruções de processamento.

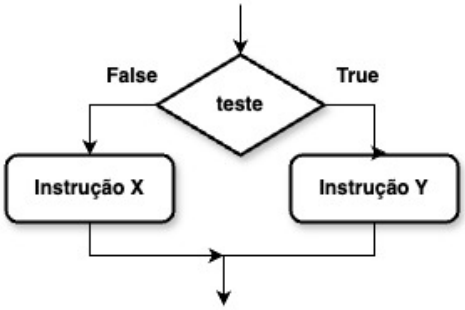



*Figura 30 - Estrutura de seleção de duas vias*

### Cálculo de paridade de um número

Vejamos um exemplo simples: Vamos testar se um número introduzido pelo utilizador, é ou não um número par. Um número é par se for divisível por dois, ou seja, se o resto da divisão desse número por dois for igual a zero. Caso contrário, não é par.

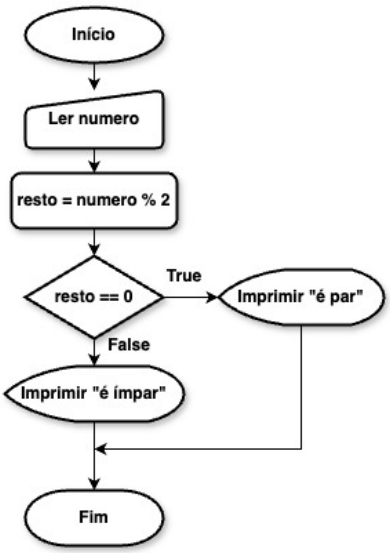



*Figura 31 - Algoritmo de teste à paridade de número*

De notar que consoante o resto da divisão, armazenado na variável `resto`, o programa segue um dos dois caminhos, ou imprime a mensagem `par` ou imprime a mensagem `ímpar`, após comparar a variável `resto` com o valor zero(`resto == 0`). Traduzindo o algoritmo para Python teríamos algo semelhante ao programa seguinte:

```
== RESTART: /Users/miguel/Desktop/Work/ESAN/Ens
ino/__book/selecao2vias_par.py ==
Insira um número: 6
é par
>>>
= RESTART: /Users/miguel/Desktop/Work/ESAN/Ensi
no/__book/selecao2vias_par.py
Insira um número: 7
é ímpar
>>>
```

```
numero=int(input("Insira um número: "))
resto=numero%2
if resto==0:
    print("é par")
else:
    print("é ímpar")
```

*Figura 32 - Programa Python do teste de paridade de um número*

Podemos verificar que neste caso, a segunda via é aberta com a instrução `else`. Enquanto o teste lógico à variável resto é feito na instrução `if`, a instrução `else` é única, sem qualquer teste. Em verdade, a ordem que estamos a enviar é de que, **se** o teste lógico é verdade, o programa imprime a mensagem "`é par`", **senão**, imprime a mensagem "`é ímpar`".

### Teste ao sinal de um número

De modo a consolidar a estrutura de seleção de duas vias, vejamos novo exercício: determinar o sinal de um número inserido pelo utilizador, é positivo ou negativo. Ou neutro. Neste exemplo, o resultado final permite que uma de três hipóteses, positivo, negativo, neutro.

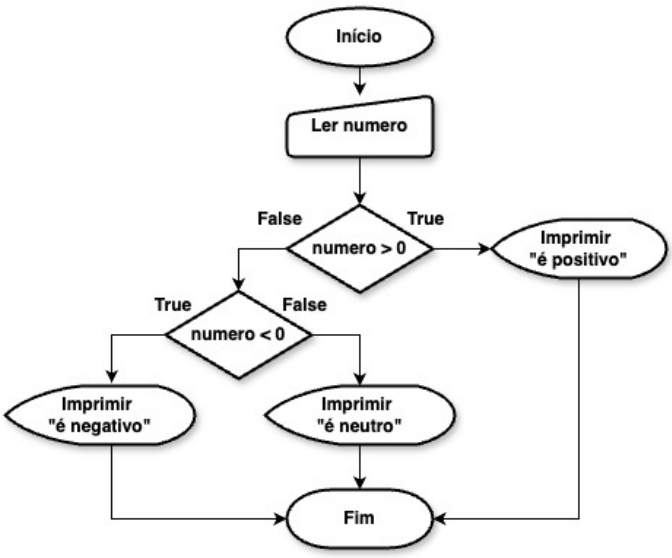



*Figura 33 - Algoritmo de teste ao sinal de um número*

Como se verifica em cima, o algoritmo imprime uma das três hipóteses, consoante o valor inserido pelo utilizador. Para este caso, como ainda só analisamos a seleção de uma e duas vias, adaptámos duas seleções de duas vias para que o programa permita as três hipóteses possíveis. Em Python seria algo assim:

```
>>>
= RESTART: /Users/miguel/Desktop/Work/ESAN/Ensino/__book/selec
ao2vias_sinal.py =
Insira um número: 8
é positivo
>>>
= RESTART: /Users/miguel/Desktop/Work/ESAN/Ensino/__book/selec
ao2vias_sinal.py
Insira um número: -44
é negativo
>>>
= RESTART: /Users/miguel/Desktop/Work/ESAN/Ensino/__book/selec
ao2vias_sinal.py
Insira um número: 0
é neutro
>>> |
```

```
numero=int(input("Insira um número: "))
if numero>0:
    print("é positivo")
else:
    if numero<0:
        print("é negativo")
    else:
        print("é neutro")
```

*Figura 34- Programa de teste de sinal de um número*

Repare que a segunda seleção de duas vias (`if/else`) está indentada no `else` da primeira seleção de duas vias, tal como no fluxograma, o que corresponde ao cenário do teste `numero<0` ser falso. De salientar ainda que os testes realizados poderiam ser outros, por exemplo `numero==0`, e dispostos noutra organização.

De qualquer forma, se o programa possibilita 3 vias, existe outro modo mais eficaz para se resolver o problema, a seleção de n vias, one **n** são o número de vias que o programador determinar serem necessárias.

### Seleção de n vias

Tal como o nome indica, uma seleção de n vias permite criar vários caminhos possíveis de execução de um algoritmo e programa e, tal como nas outras subestruturas de seleção, apenas a via validada em teste lógico será executada.

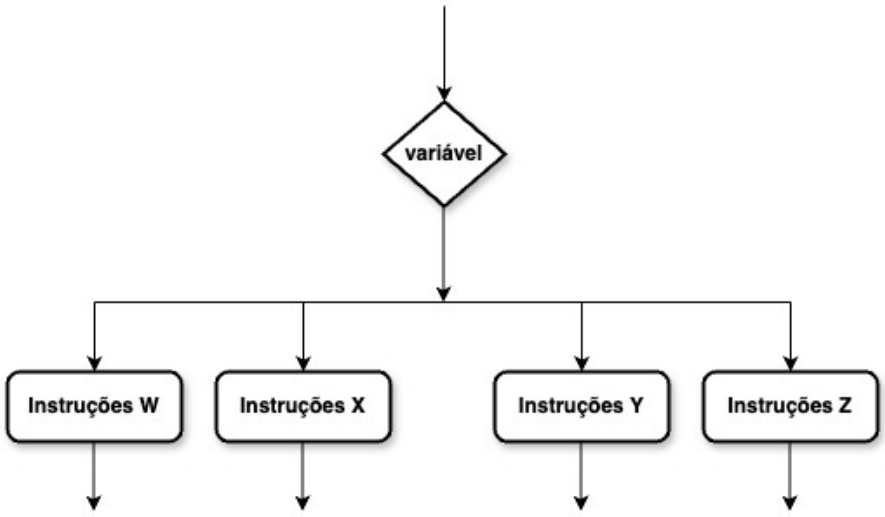



*Figura 35 - Estrutura de seleção de n vias*

No contexto do problema anterior, uma seleção de n vias, em concreto, uma seleção de três vias, é a mais adequada para o teste ao sinal de um número:

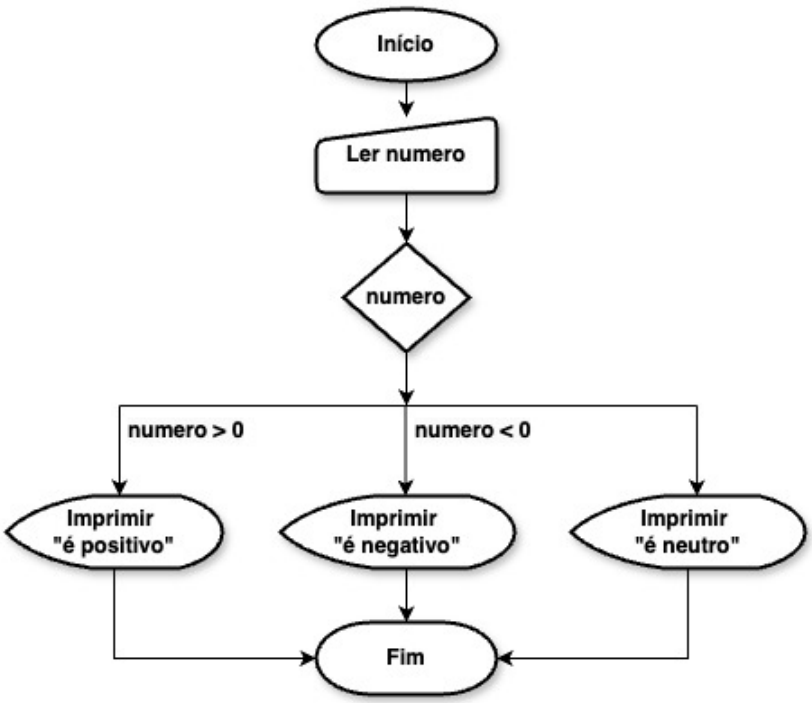



*Figura 36 - Algoritmo de teste ao sinal de um número, com seleção de 3 vias*

O fluxo de dados torna-se mais simples de visualizar. De salientar que se mantém os dois testes, `numero>0` e `numero<0`, tal como no algoritmo com duas seleção de duas vias. Neste exemplo, cada uma das vias é indexada a um teste, com exceção da última, onde é impresso que o número é neutro. Pense na última via sem teste (else) como "*nenhuma das opções anteriores*". Em Python teríamos:

```
numero=int(input("Insira um número: "))
if numero>0:
    print("é positivo")
elif numero<0:
    print("é negativo")
else:
    print("é neutro")
```

*Figura 37 - Programa de teste de sinal de um número, com seleção de 3 vias*

Numa seleção de n vias, é preciso ter sempre em mente que:

1) A primeira via, terá sempre um teste com a instrução `if`;
2) A segunda e todas as vias intermédias até à penúltima terão testes com a instrução `elif`, conjugação de `else` com `if`;
3) A última via deve ser sempre fechada com um `else`, sem teste.

### Maior e menor de dois números

Vejamos um novo exercício. Determine o maior e o menor de dois números inseridos pelo utilizador. A primeira análise que deveremos efetuar é sobre quais as possíveis soluções: o primeiro número é maior do que o segundo (1); o segundo número é maior que o primeiro (2); e ainda pode acontecer o caso dos dois números inseridos serem iguais (3) e, deste modo, mitigamos qualquer ambiguidade que poderíamos estar a criar.

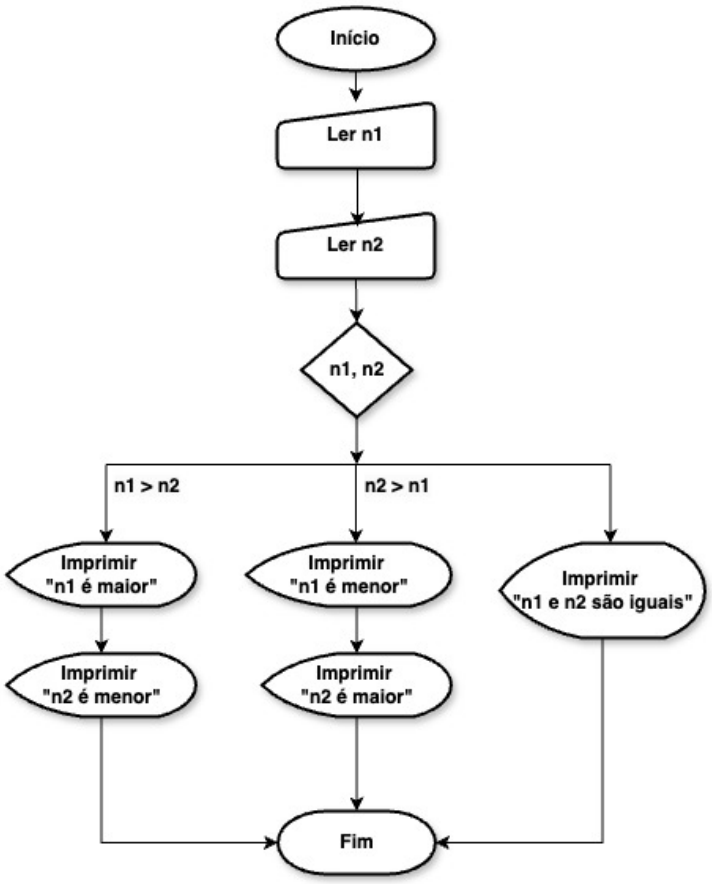



*Figura 38 - Algoritmo para determinar maior e menor de dois números*

Novamente, reiteramos que os testes lógicos e algoritmo anterior podem ser criados doutra forma. Neste exemplo, a seleção de três vias está sujeita ao teste de duas variáveis, `n1` e `n2`, razão pela qual são indicadas no símbolo de teste. Em Python, teríamos:

```
n1=int(input("Insira um número: "))
n2=int(input("Insira outro número: "))

if n1>n2:
    print("n1 é maior")
    print("n2 é menor")
elif n2>n1:
    print("n1 é menor")
    print("n2 é maior")
else:
    print("n1 e n2 são iguais")
```

*Figura 39 - Programa para determinar maior e menor de dois números*

**Observação:** A seleção de n vias é implementada em Python através das instruções `if`, `elif` e `else`. A partir da versão 3.10 em Python, uma seleção de n vias também pode ser implementada com as instruções `match` e `case` [15] embora, não será utilizada neste livro, por ser uma implementação muito recente na linguagem. Outras linguagens de programação, como C++, VB ou Java, criaram as instruções `switch` e `case`, para implementar esta subestrutura.

## Desafios

Nesta secção pretende-se que os conceitos da estrutura de seleção sejam consolidados com uma bateria de exercícios. Com diferentes exemplos verificará a vantagem da aplicação da estrutura de seleção na resolução de exercícios.

### Maior e menor de três números

O desafio que iremos apresentar pretende que o leitor construa o algoritmo e respetivo programa que determine o maior e o menor de três números. À primeira vista, parece simples e idêntico ao desafio anterior para determinar o maior e menor de dois números. No entanto, se seguir a mesma estratégia, terá de executar 9 testes para verificar todos os cenários possíveis: para números distintos, teríamos seis soluções (3!), para dois iguais (aab;aba;baa) três, e para todos iguais, apenas um.

Será esta a forma mais eficaz de resolver o desafio? Talvez não. Além disso, imagine que o problema passaria por determinar o maior e menor, não de três, mas de quatro números iguais' Ou até cinco ou mais? O número de estruturas de decisão aumentaria exponencialmente. Neste sentido, para este desafio a solução passa por nos abstrairmos dos 3 números e centrar o raciocínio no maior e no menor. A solução a apresentar servirá para uma infinidade de números, já que o algoritmo se adapta a diferentes quantidades. É um algoritmo baseado nas funções MAX() e MIN() que existem nas calculadoras e folhas de cálculo:

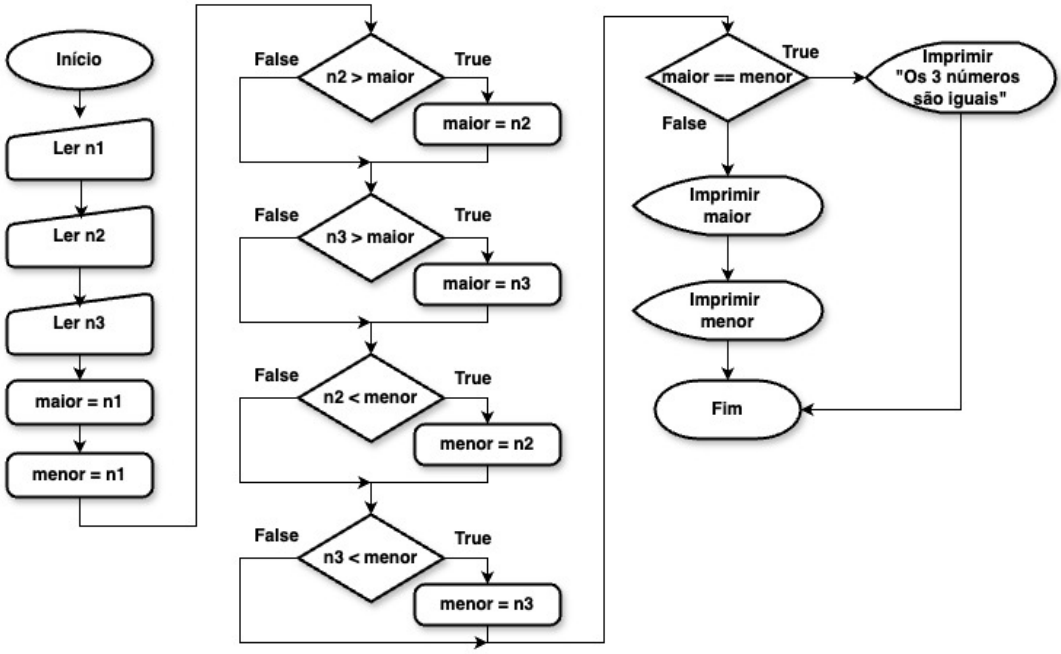



*Figura 40 - Algoritmo para determinar maior e menor de três números*

A estratégia deste algoritmo assenta na abstração em atribuir ao primeiro número inserido pelo utilizador a classificação de `maior` e `menor`. Em seguida, comparam-se os outros dois números, não com o primeiro, mas com os valores guardados nas variáveis `maior` e `menor` (quatro seleção de uma via) e atualizam-se se os valores dos números forem maiores ou menores. No final, verifica-se se as variáveis `maior` e `menor` são iguais. Se forem, os três números são iguais. Se não forem, temos pelo menos um dos números maior que o outro(s) e pelo menos um dos números que é menor que o outro(s).

```
n1=int(input("Insira um número: "))
n2=int(input("Insira outro número: "))
n3=int(input("Insira ainda outro número: "))

maior=n1
menor=n1

if n2>maior:
    maior=n2
if n3>maior:
    maior=n3
if n2<menor:
    menor=n2
if n3<menor:
    menor=n3

if maior==menor:
    print("Os 3 números são iguais")
else:
    print("maior = "+str(maior))
    print("menor = "+str(menor))
```

*Figura 41 – Programa para determinar maior e menor de três números*

### Cálculo do zero de uma equação de 1º grau – Equação da reta

Uma reta consiste numa linha reta é a distância mais curta entre dois pontos na geometria euclidiana. É um conceito fundamental em matemática, linha que se estende infinitamente em ambas as direções sem qualquer curvatura. Em notação matemática, no plano bidimensional, uma reta pode ser representada por uma equação linear, como $y=mx+b$, onde $m$ é a inclinação da reta e $b$ é a interseção com o eixo $y$ (ou ponto de origem, no caso de uma reta que passa pela origem) [13] [16].

A equação linear $y=mx+b$ é também uma equação do primeiro grau, tendo em consideração que a incógnita $x$ é elevada à primeira potência ($x^1$). Os valores $m$ e $b$, da inclinação e interseção com o eixo $y$, respetivamente, definem a estrutura da reta. Vejamos três exemplos que definem os três cenários possíveis. O primeiro cenário corresponde a uma inclinação e interseção positiva, sendo a equação $y=2x+3$:

| | |
|---|---|
| **m** | **2** |
| **b** | **3** |
| | |
| **x** | **y = mx + b** |
| -5 | -7 |
| -4 | -5 |
| -3 | -3 |
| -2 | -1 |
| -1 | 1 |
| 0 | 3 |
| 1 | 5 |
| 2 | 7 |
| 3 | 9 |
| 4 | 11 |
| 5 | 13 |

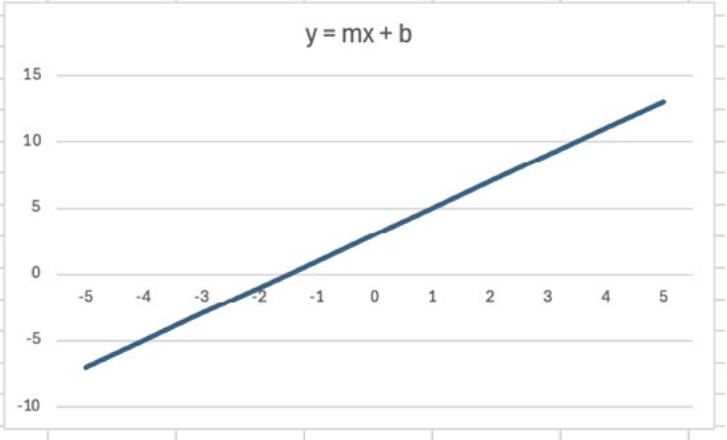



*Figura 42 - Gráfico da reta y=2x+3*

Outro exemplo, com inclinação negativa, $y=-5x+4$:

| | |
|---|---|
| m | -5 |
| b | 4 |
| | |
| x | y = mx + b |
| -5 | 29 |
| -4 | 24 |
| -3 | 19 |
| -2 | 14 |
| -1 | 9 |
| 0 | 4 |
| 1 | -1 |
| 2 | -6 |
| 3 | -11 |
| 4 | -16 |
| 5 | -21 |

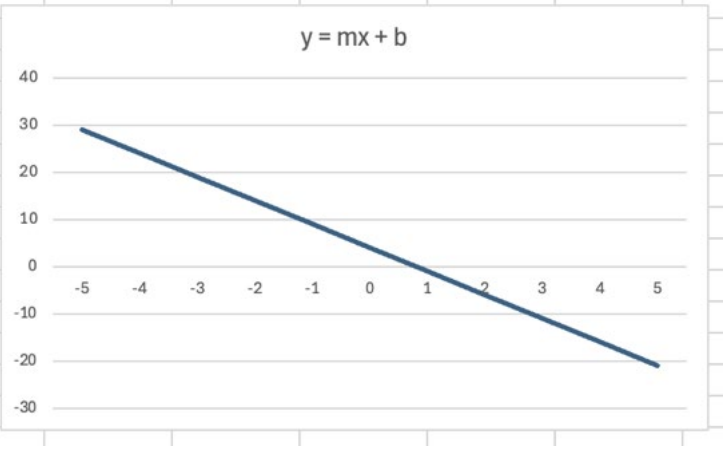



*Figura 43 - Gráfico da reta y=-5x+4*

E, por último, sem inclinação, ou seja, $x=0$, implicando, por exemplo, $y=4$:

| | |
|---|---|
| m | 0 |
| b | 4 |
| | |
| x | y = mx + b |
| -5 | 4 |
| -4 | 4 |
| -3 | 4 |
| -2 | 4 |
| -1 | 4 |
| 0 | 4 |
| 1 | 4 |
| 2 | 4 |
| 3 | 4 |
| 4 | 4 |
| 5 | 4 |

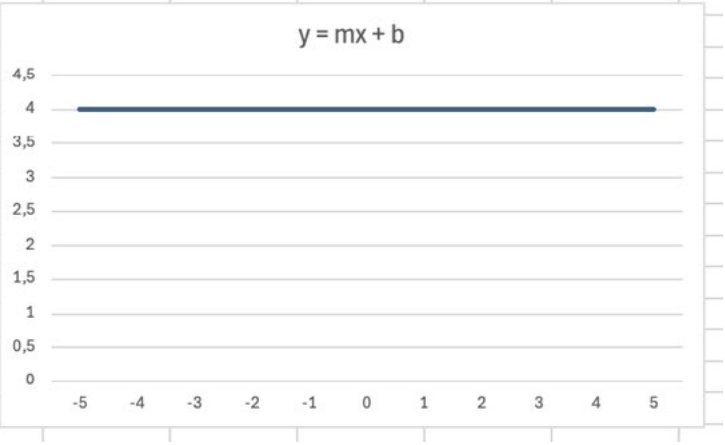



*Figura 44 - Gráfico da reta y=4*

Desde que a reta tenha inclinação, ou seja, $m\neq0$, a reta interseta sempre o eixo $y$. Esse ponto de interseção com o eixo $y$ é conhecido como o zero da reta o ou zero da equação. Quando a reta não tem declive, $m=0$, nunca interseta o eixo $y$.

O desafio que propomos, é o seguinte: dada uma inclinação $m$ e um declive $b$, definidos pelo utilizador, calcule o zero da reta, $x$, caso ele exista. Numa primeira análise podemos concluir que caso a inclinação $m$, definida pelo utilizador, seja igual a zero, a reta não tem zero. Caso contrário, é determinado igualando a zero a equação:

$$mx + b = 0$$
$$mx = -b$$
$$x = -b/m$$

Com o exemplo da *Figura 42 - Gráfico da reta y=2x+3*, o cálculo seria:

$$2x + 3 = 0$$
$$2x = -3$$
$$x = -3/2$$
$$x = -1.5$$

Verificando no gráfico, quando $x=-1.5$, a reta interseta o eixo $y$ sendo esse ponto o zero da reta $y=2x+3$. Ou seja, $x=-b/m$ permite calcular o zero da reta, desde que haja inclinação, ou seja, este cálculo só poderá ser realizado se o utilizador tiver inserido um valor $m$ diferente de $0$. Por exemplo, na equação da *Figura 44 - Gráfico da reta y=4*, implicaria que o utilizador tivesse inserido uma inclinação $m=0$ e uma interseção $b=4$. Neste caso, o cálculo do zero x=-b/m não faria sentido. Geometricamente porque observando o gráfico, a reta y=4 nunca interseta o eixo y. Matematicamente teríamos:

$$-\frac{4}{0} \to \infty$$

(qualquer divisão por zero tende para infinito, solução impossível)

Em programação resultaria no erro de execução **`ZeroDivisionError: division by zero`**. Lembre-se de que os computadores são "bichos matemáticos", que não sabem dividir por zero! O algoritmo para calcular o zero de uma equação linear da reta seria:

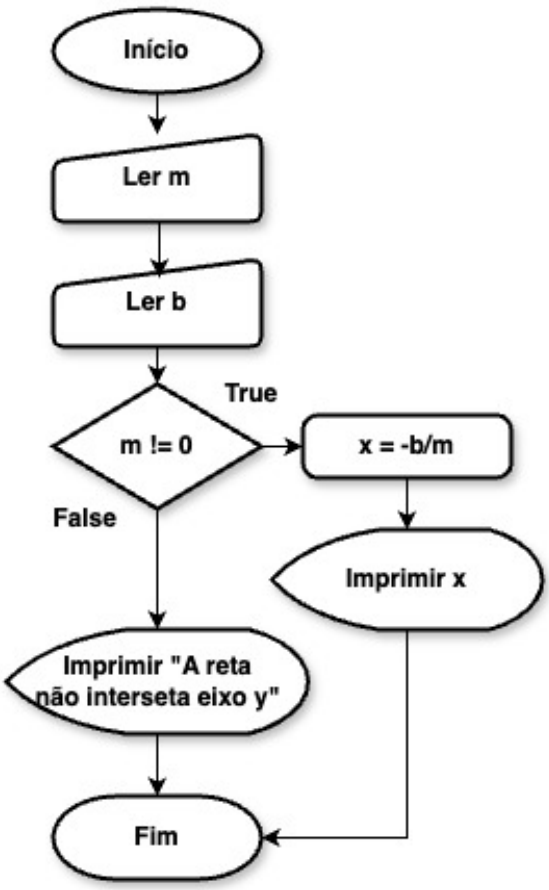



*Figura 45 - Algoritmo do cálculo do zero de uma reta*

Repare que a divisão `x=-b/m` só é realizada quando temos a garantia de que a variável `m` é diferente de zero.

```
m=int(input("Insira a inclinação m: "))
b=int(input("Insira a interseção b: "))

if m!=0:
    x=-b/m
    print("O zero da reta é "+str(x))
else:
    print("A reta não interseta o eixo y")
```

*Figura 46 - Programa para determinar o zero de uma reta*

### Cálculo dos zeros de uma equação de 2º grau – Equação da parábola

Uma parábola é uma curva plana e simétrica que pertence à família das cónicas, sendo definida como o lugar geométrico de pontos que estão à mesma distância de um ponto fixo (chamado de foco) e de uma reta fixa (chamada de diretriz) [14]. Em notação matemática, no plano bidimensional, uma parábola pode ser representada por uma equação quadrática do segundo grau ($x$ à segunda potência), como $y=ax^2+bx+c$, onde $a$, $b$ e $c$ são coeficientes reais e, em particular, o coeficiente $a$ corresponde à orientação da concavidade da parábola, sendo virada para cima se $a>0$, ou virada para baixo se $a<0$.

Vejamos três exemplos que definem os três cenários possíveis. O primeiro cenário corresponde a uma parábola com concavidade para cima, que interseta o eixo $y$ uma vez em $x=-1$, ou seja, a parábola tem um zero com a equação $y=x^2+2x+1$. O(s) zero(s) é a definição dada a cada interseção da parábola com o eixo $y$:

| | |
|---|---|
| a | 1 |
| b | 2 |
| c | 1 |
| | |
| x | y = ax2+bx+c |
| -10 | 81 |
| -9 | 64 |
| -8 | 49 |
| -7 | 36 |
| -6 | 25 |
| -5 | 16 |
| -4 | 9 |
| -3 | 4 |
| -2 | 1 |
| -1 | 0 |
| 0 | 1 |
| 1 | 4 |
| 2 | 9 |
| 3 | 16 |
| 4 | 25 |
| 5 | 36 |
| 6 | 49 |
| 7 | 64 |
| 8 | 81 |
| 9 | 100 |
| 10 | 121 |

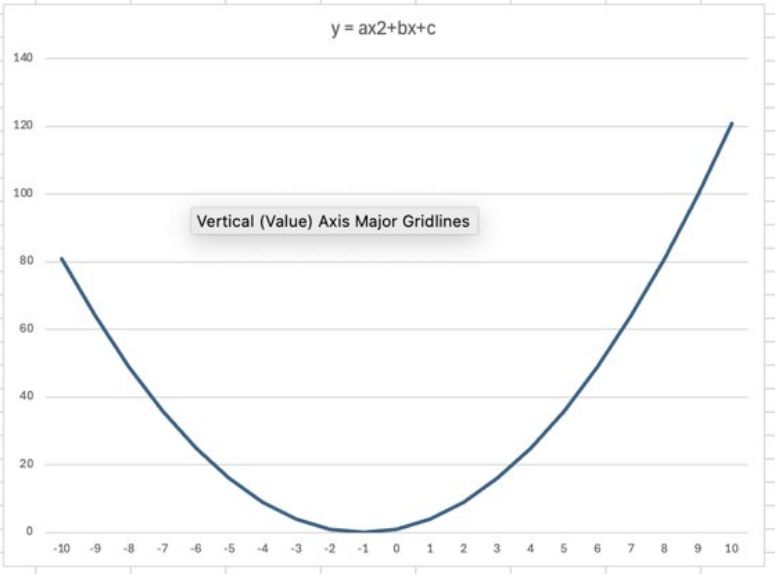



*Figura 47 – Gráfico da parábola $y=x^2+2x+1$*

Parábola $y=-x^2+6x+1$, com concavidade para baixo e interseção no eixo $y$ em dois pontos (dois zeros)

| | |
|---|---|
| a | -1 |
| b | 6 |
| c | 1 |

| x | y = ax2+bx+c |
|---|---|
| -10 | -159 |
| -9 | -134 |
| -8 | -111 |
| -7 | -90 |
| -6 | -71 |
| -5 | -54 |
| -4 | -39 |
| -3 | -26 |
| -2 | -15 |
| -1 | -6 |
| 0 | 1 |
| 1 | 6 |
| 2 | 9 |
| 3 | 10 |
| 4 | 9 |
| 5 | 6 |
| 6 | 1 |
| 7 | -6 |
| 8 | -15 |
| 9 | -26 |
| 10 | -39 |

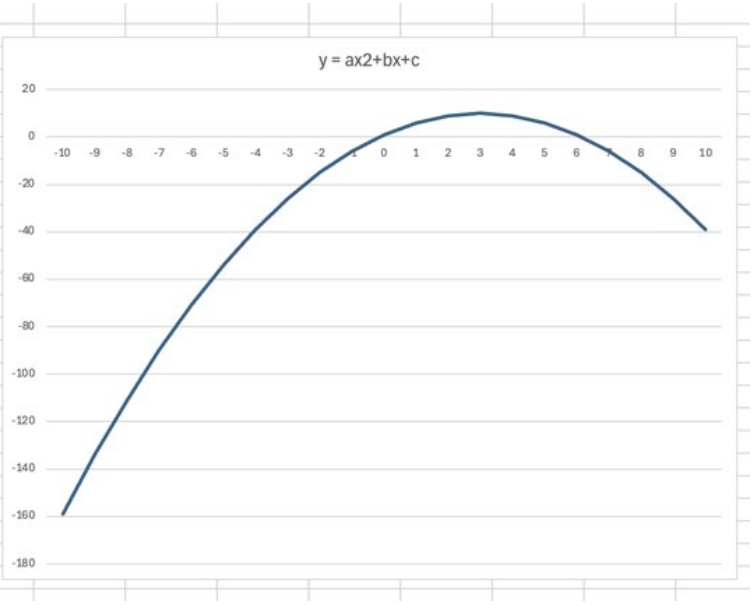



*Figura 48- Gráfico da parábola $y=-x^2+6x+1$*

Parábola $y=-4x^2+x+40$, com concavidade para cima sem interseção no eixo $y$ (sem zeros)

| | |
|---|---|
| a | 4 |
| b | 1 |
| c | 40 |

| x | y = ax2+bx+c |
|---|---|
| -10 | 430 |
| -9 | 355 |
| -8 | 288 |
| -7 | 229 |
| -6 | 178 |
| -5 | 135 |
| -4 | 100 |
| -3 | 73 |
| -2 | 54 |
| -1 | 43 |
| 0 | 40 |
| 1 | 45 |
| 2 | 58 |
| 3 | 79 |
| 4 | 108 |
| 5 | 145 |
| 6 | 190 |
| 7 | 243 |
| 8 | 304 |
| 9 | 373 |
| 10 | 450 |

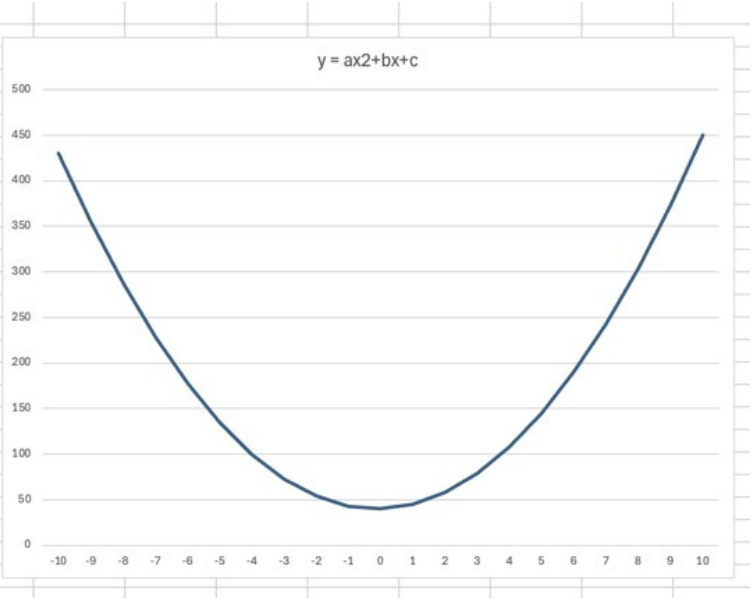



*Figura 49 - Gráfico da parábola $y=4x^2+x+40$*

O desafio que propomos ao leitor é o seguinte: Dados os coeficientes $a$, $b$ e $c$, introduzidos pelo utilizador, determine se a parábola interseta, ou não, ou eixo $y$. Caso intersete, calcule o(s) zero(s).

Numa primeira na análise devemos verifica se o coeficiente $a$ é diferente de zero pois, se tal não se verificar, teremos uma equação da reta com $y=bx+c$. Ultrapassada esta dificuldade, temos de verificar se a parábola interseta o eixo $y$ em duas vezes (dois zeros), uma vez (um zero) ou nenhuma vez. A forma mais rápida para ser fazer esta verificação será através da fórmula resolvente que é dada por:

Voltando ao caso específico do a=0, repare que não é possível utilizar a fórmula porque estaremos novamente no problema da divisão de algo por zero:

A formula resolvente dá-nos a chave para determinar se a parábola tem dois, um ou zero zeros, estando o "segredo" no conteúdo $b^2-4ac$ da raiz quadrada. Para simplificar trataremos $b^2-4ac$ por $\Delta$:

- Se $\Delta>0$, teremos uma raiz quadrada positiva, logo, dois zeros, $x1$ e $x2$, correspondentes a duas interseções no eixo $y$;
- Se $\Delta=0$, teremos uma raiz quadrada , correspondendo a um zero, $x$, correspondente a uma interseção no eixo $y$;
- Se $\Delta<0$, a raiz quadrada é negativa e não solução no espaço real $\mathbb{R}$, não tem zeros, logo, a parábola não interseta o eixo $y$.

O algoritmo seria:

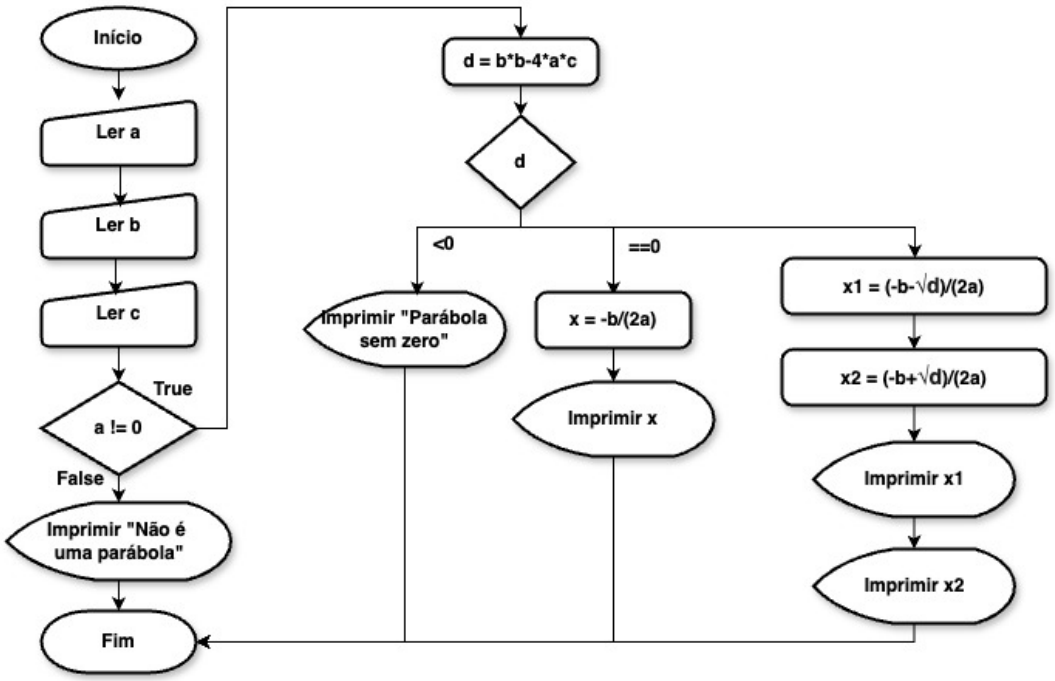



*Figura 50 - Algoritmo de cálculo dos zeros de uma parábola*

Antes de apresentarmos o programa em Python, vamos só introduzir algumas considerações quanto a potências e raízes, necessárias para a execução do programa. No algoritmo anterior, colocamos `b*b`, embora pudéssemos ter feito `b**2` que em Python corresponde a $b^2$(b à potência de dois).

Na via em que é necessário calcular a raiz quadrada de $b^2-4ac$, podemos fazer de uma das maneiras. Em matemática, pode ser calculado como, em Python, poderemos também executar a mesma operação de ambas as formas, uma delas, com recurso à função `sqrt()` da biblioteca `math` do Python:

```
import math

d=math.sqrt(d*d-4*a*c)
```

Ou poderíamos executar a mesma instrução com recurso à potência de `1/2`:

```
d = (b*b-4*a*c)**(1/2)
```

Ambas as opções são válidas. No entanto, para o leitor se familiarizar coma utilização de bibliotecas em Python, onde existem centenas para diferentes fins e propósitos, iremos utilizar a função `sqrt()` da biblioteca `math`. A tradução do algoritmo em Python seria:

```
import math  #importação da bibloteca math

a=int(input("insira coeficiente a: "))
b=int(input("insira coeficiente b: "))
c=int(input("insira coeficiente c: "))

if a!=0:
    d=b*b-4*a*c
    if d<0:
        print("Parábola sem zero")
    elif d==0:
        x=-b/(2*a)
        print("x = "+str(x))
    else:
        x1=(-b-math.sqrt(d))/(2*a)
        x2=(-b-math.sqrt(d))/(2*a)
        print("x1 = "+str(x1))
        print("x2 = "+str(x2))
else:
    print("Não é uma parábola!")
```

*Figura 51 - Programa de cálculo dos zeros de uma parábola*

### Calculadora aritmética simples

Uma calculadora aritmética simples é um dispositivo ou programa que realiza operações matemáticas básicas, como:

- **Adição (+):** Soma dois números.
- **Subtração (-):** Subtrai um número de outro.
- **Multiplicação (×):** Multiplica dois números.
- **Divisão (÷):** Divide um número por outro.

Ela é usada para cálculos do dia a dia e não inclui funções avançadas, como raiz quadrada, exponenciação ou operações trigonométricas, que são comuns em calculadoras científicas. Pode ser física (como uma calculadora de bolso) ou digital (como um programa). O desafio que agora propomos ao leitor é o de criar um algoritmo e programa, que atue como uma calculadora aritmética simples. O programa inicia pedindo ao utilizador o primeiro operando, a operação (+,-,*,/), o segundo operando e, por fim, apresenta o resultado da operação.

Devemos ter atenção à seguinte ambiguidade que resulta num erro de execução ou num erro semântico: Se pretendermos realizar uma divisão de um número por zero, temos de prevenir que o programa termina – lembre-se que qualquer divisão por zero é impossível de realizar e, matemáticamente, o seu resultado tende para infinito. Como devemos proceder? Nas calculadoras aritméticas físicas, se tentar dividir algo por zero, a calculadora apresenta no visor 0. Apesar deste resultado ser um erro científico, os fabricantes entenderam ser mais simples para a maioria dos utilizadores apresentar um 0 ao invés de imprimir um . Pretende-se manter o mesmo padrão no nosso algoritmo. Uma última validação: prevenir o utilizador de inserir um operando inválido que não corresponde a nenhuma operação aritmética simples, por exemplo, `$`:

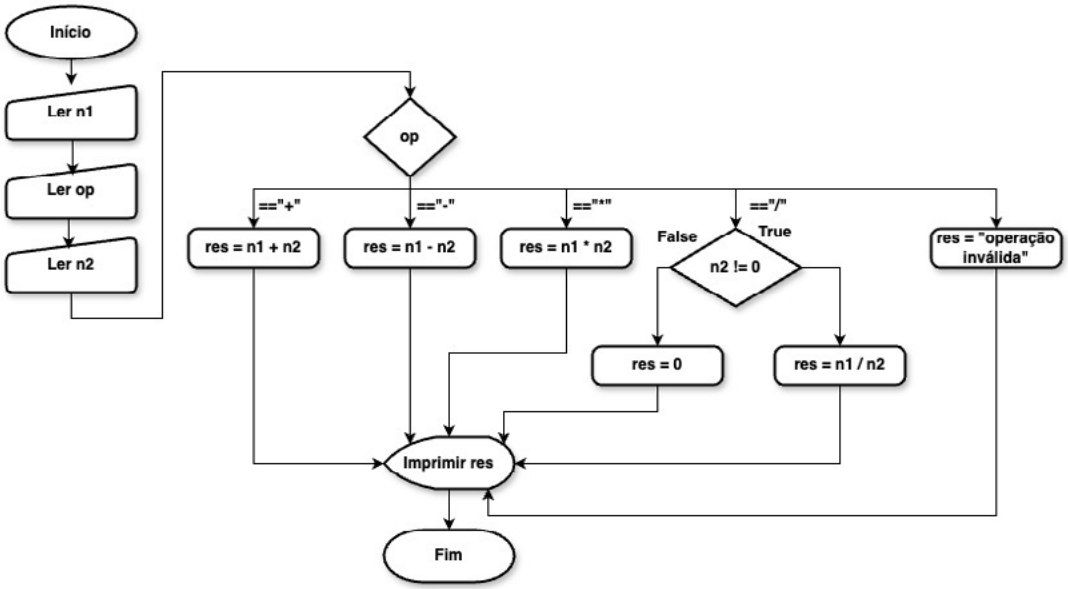



*Figura 52 - Algoritmo de uma calculadora aritmética simples*

```
n1=float(input("Primeiro operando: "))
op=input("Operação (+,-,*,/): ")
n2=float(input("Segundo operando: "))

if op=="+":
    res=n1+n2
elif op=="-":
    res=n1-n2
elif op=="*":
    res=n1*n2
elif op=="/":
    if n2==0:
        res=0
    else:
        res=n1/n2
else:
    res="Operação inválida"

print(res)
```

*Figura 53 - Programa de uma calculadora científica*

No programa anterior introduzimos uma alternativa no `input` dos operandos. Convertemos o `input` em `float()`, de modo que os operandos inseridos mantenham o seu valor decimal. A função `float()` vem de ponto flutuante que representa os números reais com parte inteira e parte fracionária.

Deixamos ainda uma sugestão: Crie o algoritmo e programa que, após receber do utilizador os coeficientes a, b e c, pelo utilizador, indique se é uma reta ou uma parábola, e se há interseções com o eixo y, e quais (zero(s) equação).

### Cálculo do Índice de Massa Corporal (IMC)

O Índice de Massa Corporal (IMC) é uma medida utilizada para classificar de forma simples se uma pessoa se encontra num intervalo saudável do seu peso em relação à sua altura. Adotada pela Organização Mundial de Saúde, a fórmula é definida por:

$$IMC = \frac{peso\ (Kg)}{altura^2(m^2)}$$

E a tabela de interpretação do IMC é definida da seguinte forma:

| IMC | Classificação |
|---|---|
| < 16 | Magreza grave |
| 16 a < 17 | Magreza moderada |
| 17 a < 18,5 | Magreza leve |
| 18,5 a < 25 | Saudável |
| 25 a < 30 | Sobrepeso |
| 30 a < 35 | Obesidade Grau I |
| 35 a < 40 | Obesidade Grau II (severa) |
| > 40 | Obesidade Grau III (mórbida) |

*Tabela 5 – Classificação do peso em função de intervalos de valor IMC*

Apesar de ser uma medida útil e simples, o IMC não leva em consideração fatores como a composição corporal (proporção de músculo e gordura), portanto, não é uma avaliação completa da saúde. Por exemplo, uma pessoa muito musculosa pode ter um IMC elevado, mas não ter excesso de gordura corporal. Mas para este desafio, iremos utilizar esta classificação e fórmula. O desafio consiste em calcular o IMC em função do peso (em Kilos) e altura (em metros) inseridas pelo utilizador. Após a apresentação do valor IMC, pretende-se informar o utilizador da classificação obtida, de acordo com a Tabela 5:

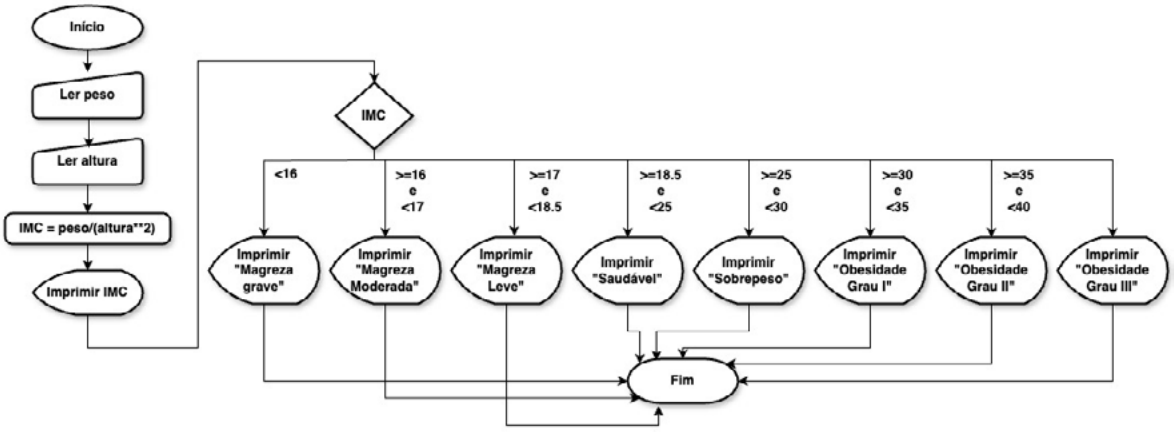



*Figura 54 - Algoritmo de cálculo do IMC e classificação*

Após o *input* do peso e da altura, calculamos o IMC e criamos uma seleção de n vias, para cada intervalo de classificação. Nos testes lógicos cujo IMC está compreendido entre um intervalo, balizamos a fronteira do intervalo com o operador lógico e (em Python, `and`). Em Python o programa será:

```
peso=float(input("Indique o peso em Kg : "))
altura=float(input("Indique a altura em metros : "))

IMC=peso/(altura**2)

print("IMC = "+str(IMC)+" Kg/m2")

if IMC<16:
    print("Magreza grave")
elif IMC>=16 and IMC<17:
    print("Magreza moderada")
elif IMC>=17 and IMC<18.5:
    print("Magreza leve")
elif IMC>=18.5 and IMC<25:
    print("Saudável")
elif IMC>=25 and IMC<30:
    print("Sobrepeso")
elif IMC>=30 and IMC<35:
    print("Obesidade Grau I")
elif IMC>=35 and IMC<40:
    print("Obesidade Grau II (severa)")
else:
    print("Obesidade Grau III (mórbida)")
```

*Figura 55 - Programa de cálculo do IMC e classificação*

### Cálculo de ano bissexto

Vamos realizar um desafio que utilize os operadores lógicos `and` e `or` para verificar se um ano é bissexto ou não. Diz a definição que todos os anos que sejam múltiplos de 4 mas que não sejam múltiplos de 100, com exceção daqueles que são múltiplos de 400, são bissextos.

O que na realidade acontece é que o planeta Terra demora aproximadamente 365,2422 dias solares (1 ano trópico) para dar uma volta completa ao redor do Sol, enquanto o ano-calendário comum (por convenção) tem 365 dias solares. Sobram, portanto, aproximadamente 5h48m46s (0,2422 dia) a cada ano trópico. As horas excedentes são somadas e adicionadas ao calendário na forma inteira de um dia (4 x 6h = 1 dia), razão pela qual de quatro em quatro anos acrescentamos um dia ao calendário [17].

Como 5h48m46s não são 6horas, só de 400 em 400 anos é que consideramos anos bissextos os inícios de século, para acertar os 11m14s que demos a mais a cada ano bissexto, durante os quatrocentos anos.

O algoritmo deve verificar que um ano é bissexto se esse ano é divisível por 4 e não por 100. Ou então ou ano é divisível por 400. Um dos dois testes valida o ano bissexto. Uma das soluções passaria por executar seleção de duas ou de n vias. Porém, é possível numa única seleção de duas vias, com recurso aos operandos lógicos, executar todos os testes:

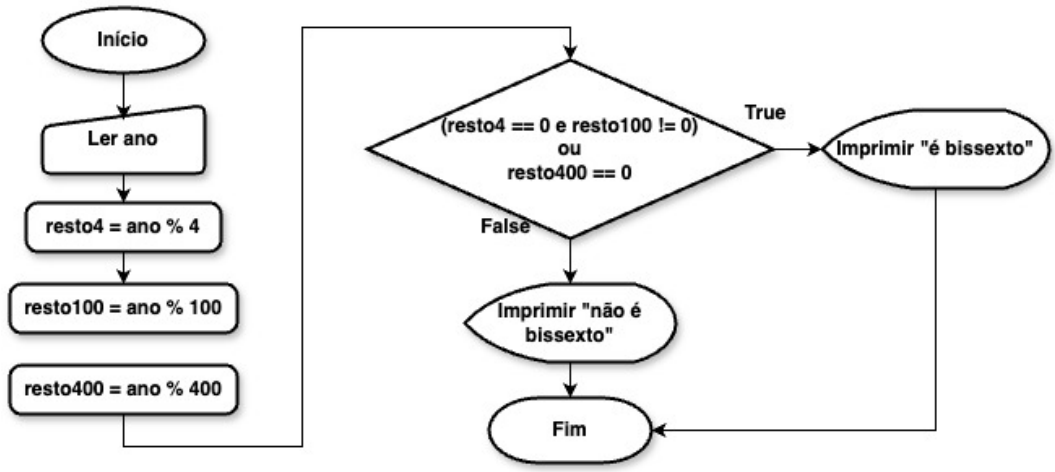



*Figura 56 - Algoritmo de cálculo de ano bissexto*

Traduzindo em Python o programa será:

```
ano=int(input("Indique o ano : "))

resto4=ano%4
resto100=ano%100
resto400=ano%400

if (resto4==0 and resto100!=0) or resto400==0:
    print(str(ano)+" é bissexto!")
else:
    print(str(ano)+" não é bissexto!")
```

*Figura 57 - Programa de cálculo de ano bissexto*

## Repetição

A última estrutura de controlo a abordar é designada por repetição. Esta estrutura permite que o fluxo do programa se repetia mediante um determinado teste lógico (`True` ou `False`) ou de acordo com o número de repetições, conhecidas por itera-

ções, definidas. A repetição pode ser condicional, com o teste lógico à cabeça das instruções a repetir, ou com o teste lógico à cauda das instruções que se pretendem repetir. Podemos ainda definir um número específico de ciclos de repetição.

### Repetição com teste à cabeça

Numa repetição com teste à cabeça, há a necessidade de tomar uma decisão com base no valor lógico duma expressão. No entanto, a mesma ação será executada repetidamente enquanto o resultado da expressão lógica se mantiver verdadeiro (`True`).

O teste (da expressão lógica) precede a ação. Diz-se, por isso, que o teste é à cabeça, e funciona como uma condição de paragem (`false`) dos ciclos ou repetições.

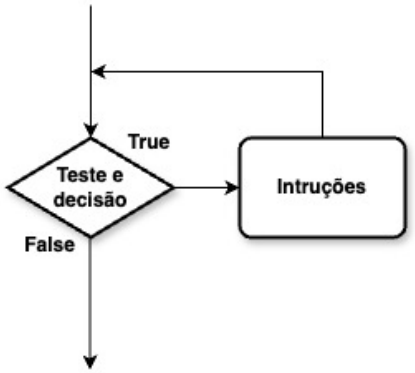



*Figura 58 – Estrutura de repetição com teste à cabeça*

### Cálculo de anos e de alturas

Vejamos um exercício simples para ajudar entender a repetição com teste à cauda. Dois irmãos, João e Pedro têm alturas diferentes, O João tem 1,50 metros e cresce em média 2 centímetros por ano, ao passo que o Pedro tem 1,45 metros mas cresce em média 3 centímetros por ano. Construa um algoritmo e respetivo código Python que calcule quantos anos o Pedro precisa para ficar mais alto que o João.

Analisando o problema poderemos manualmente elaborar uma tabela onde contabilizamos os centímetros de cada um e incrementamos o número de anos:

| João | Pedro | Nº anos |
|---|---|---|
| 150 | 145 | 0 |
| 152 | 148 | 1 |
| 154 | 151 | 2 |
| 156 | 154 | 3 |
| 158 | 157 | 4 |
| 160 | 160 | 5 |
| 162 | 163 | 6 |

*Tabela 6 – Iterações de cálculo de alturas e anos do João e do Pedro*

A cada ano que passa, são incrementados centímetros à altura do João e do Pedro, dois e três centímetros, respetivamente. Cada vez que há um incremento de alturas,

incrementamos o número de anos. Se verificar, existe um padrão de ações que se repete. Ou seja, podemos criar um conjunto de instruções que se repete, até que a altura do Pedro seja maior que a altura do João:

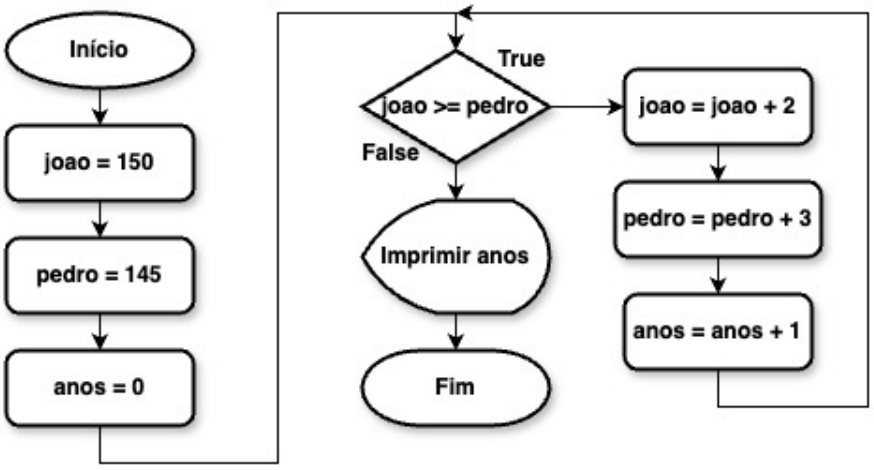



*Figura 59 - Algoritmo de cálculo de anos e alturas*

Tradução em Python:

```
joao=150
pedro=145
anos=0

while joao>=pedro:
    joao=joao+2
    pedro=pedro+3
    anos=anos+1

print("Serão necessários "+str(anos)+" anos.")
```

*Figura 60 - Programa de cálculo de alturas e anos*

### Repetição com teste à cauda

A repetição com teste à cauda é em tudo idêntica à anterior. A diferença é que o teste é feito após o processamento da ação, isto é, o teste (da expressão lógica) sucede a ação, razão pela qual se designa por repetição com teste à cauda.

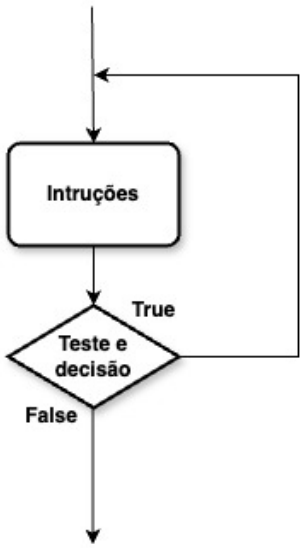



*Figura 61 - Estrutura de repetição com teste à cauda*

Enquanto numa repetição de teste à cabeça a iteração ou instruções a executar podem nunca ser executadas, caso o teste lógico não se verifique, numa repetição de teste à cauda, pelo menos uma iteração irá acontecer. Esta é a principal diferença entre as duas estruturas de repetição.

### Série de Fibonacci

Leonardo Fibonacci foi um matemático italiano nomeado como o primeiro grande matemático europeu da Idade Média. É considerado por alguns como o mais talentoso matemático ocidental da Idade Média [18]. Fibonacci ficou conhecido por utilizar uma sequência numérica que, apesar de já ser conhecida na antiguidade, foi utilizada pelo matemático numa perspetiva que estimar o crescimento da população de coelhos em 1202.

Esta sequência é também conhecida pela série da natureza, tendo em consideração a relação intrínseca que tem com assuntos como a formação de espirais e galáxias, número de pétalas de uma flor, número de ramos de uma árvore, proporções faciais, etc.. É muito utilizada para estimativas em mercados financeiros, teoria de jogos, previsão de surtos, etc..

A sequência (infinita) é composta pelos seguintes números:

0,1, **1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377,...**

Sendo que os dois números iniciais são sempre o zero (0) e o um (1). A soma destes dois números resulta no número seguinte. Todos os restantes números são resultado da soma anterior com o operando da última soma:

**0**+**1**=**1**,1+1=**2**,1+2=**3**,2+3=**5**,3+5=**8**,5+8=**13**,8+13=**21**,13+21=**34**,...

O desafio que a propor é o seguinte: Crie a série de Fibonacci até um limite definido previamente pelo utilizador, tendo em consideração que o limite mínimo será 1 (0, 1, 1):

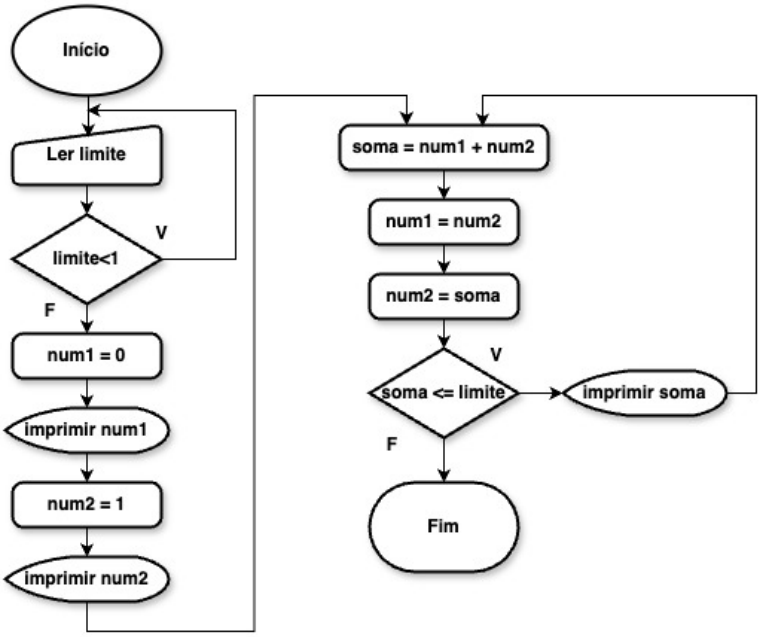



*Figura 62 – Algoritmo de apresentação da Série de Fibonacci até um limite*

Se reparar, este algoritmo apresenta duas repetições com teste à cauda: a primeira repetição, obriga o utilizador a inserir um limite válido, superior ou igual a um. A segunda repetição, atualiza os operandos e imprime a última soma, desde que esta esteja compreendida no limite definido anteriormente. A tradução em Python seria:

```
while True:
    limite=int(input("Insira o limite para gerar Fibonnacci : "))

    if limite<1:
        continue
    else:
        break

num1=0
print(num1)
num2=1
print(num2)

while True:
    soma=num1+num2
    num1=num2
    num2=soma

    if soma<=limite:
        print(soma)
    else:
        break
```

*Figura 63 - Programa de apresentação da Série de Fibonacci até um limite*

### Repetição teste à cabeça vs teste à cauda

Em termos de programação Python, a diferença de codificação de uma repetição com teste à cabeça e teste à cauda acontece na intrução `while`. No teste à cabeça, o teste é definido na própria instrução `while`:

```
while (teste lógico):
    instrução 1
    instrução 2
    .
    .
    .
    instrução n
```

*Figura 64 – Exemplo demonstrativo de construção da repetição com teste à cabeça (algoritmo da Figura 58)*

No teste à cauda, é criado um ciclo infinito com `while True:`, colocando no fim das instruções a repetir uma seleção de uma ou duas vias, permitindo a quebra do ciclo (`break`) ou a sua continuidade (`continue`):

```
while True:
    instrução 1
    instrução 2
    .
    .
    .
    instrução n

    if (teste lógico):
        break
    else:
        continue
```

*Figura 65 - Exemplo demonstrativo de construção da repetição com teste à cauda ( algoritmo da Figura 61)*

### Repetição com número pré-definido de ciclos

A repetição é idêntica à repetição com teste à cabeça embora, a diferença é que logo à partida é especificado o número de ciclos (ou iterações) que serão executados ( número de vezes que a ação será processada).

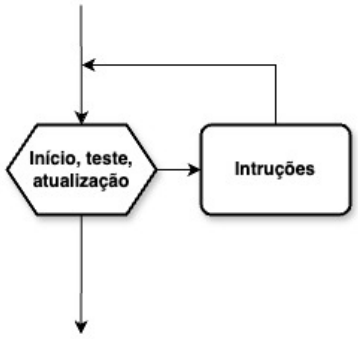



*Figura 66 - Estrutura de repetição com número pré-definido de ciclos*

## Tabuada

Como deverá recordar, uma tabuada é uma tabela matemática usada para apresentar os resultados da multiplicação de um conjunto de números, geralmente dez produtos de um número por dez unidades, de um a dez. Por exemplo a tabuada dos 7 será:

7 x 1 = 7
7 x 2 = 14
7 x 3 = 21
7 x 4 = 28
7 x 5 = 35
7 x 6 = 42
7 x 7 = 49
7 x 8 = 56
7 x 9 = 63
7 x 10 = 70

Pretende-se apresentar a tabuada de um número definido pelo utilizador. Repare que o padrão de repetição são sempre dez multiplicações. Ou seja, sabemos o número de ciclos necessários para apresentação de uma tabela de dez multiplicações:

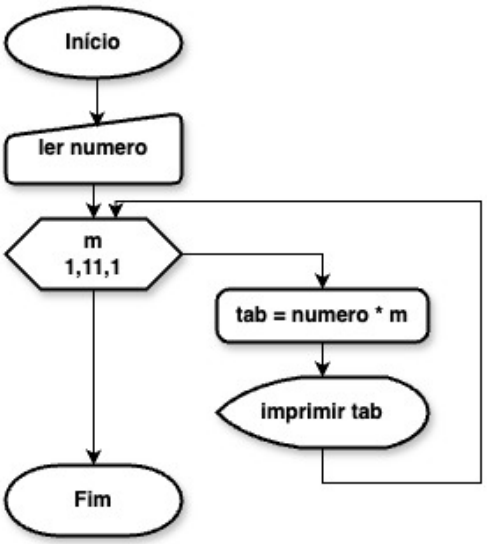



*Figura 67 – Algoritmo de geração da tabuada de um número, definido pelo utilizador*

Na repetição, criamos uma variável multiplicadora `m`, que irá variar de `1` até `<11` e que, em cada iteração, é incrementada em uma unidade.

```
numero = int(input("Indique a tabuada a apresentar : "))

for m in range(1,11,1):
    tab=numero*m
    print(str(tab)+"x"+str(numero)+"="+str(tab))
```

*Figura 68 – Programa de geração da tabuada de um número, definido pelo utilizador*

## Desafios

Seguindo a estratégia análoga da estrutura de decisão, nesta secção pretende-se que os conceitos da estrutura de repetição sejam consolidados com uma bateria de exercícios. Com diferentes exemplos verificará a vantagem da aplicação da estrutura de repetição na resolução de exercícios. Lembre-se, existem diversas soluções para os problemas apresentados. Nesta secção, a cada desafio, apresentamos uma de muitas soluções possíveis.

### Máximo Divisor Comum (MDC) e Mínimo Múltiplo Comum (mmc)

De uma forma muito simples e resumida, o Máximo Divisor Comum é o maior número que divide dois ou mais números sem deixar resto. Vejamos um exemplo:

Nomenclatura

dividendo | divisor

resto quociente

O máximo divisor comum entre 2 números, por exemplo, 348 e 156?

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 348 | \|156 | | 156 | \|36 | | 36 | \|12 | |
| 36 | 2 | | 12 | 4 | | 0 | | 3 |

**MDC (348,156) = 12**

O máximo divisor comum entre 2 números, por exemplo, 30 e 18?

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 30 | \|18 | | 18 | \|12 | | 12 | \|6 | |
| 12 | 1 | | 6 | 1 | 0 | 2 | | |

**MDC (30,18) = 6**

O mínimo múltiplo comum entre dois números, por exemplo, 12 e 18?

Multiplos de 12 são **12, 24, 36, 48, ...**
Multiplos de 18 são **18, 36, 54, ...**
O menor número comum é **36.**
**mmc(12,18) = 36**

O **mmc** também se pode calcular através da fórmula

O desafio proposto é o cálculo do MDC e do mmc de dois números. Deve ter em consideração que todos os números naturais, são passíveis de cálculo do MDC e mmc e que o MDC e mmc de dois números iguais são o próprio número:

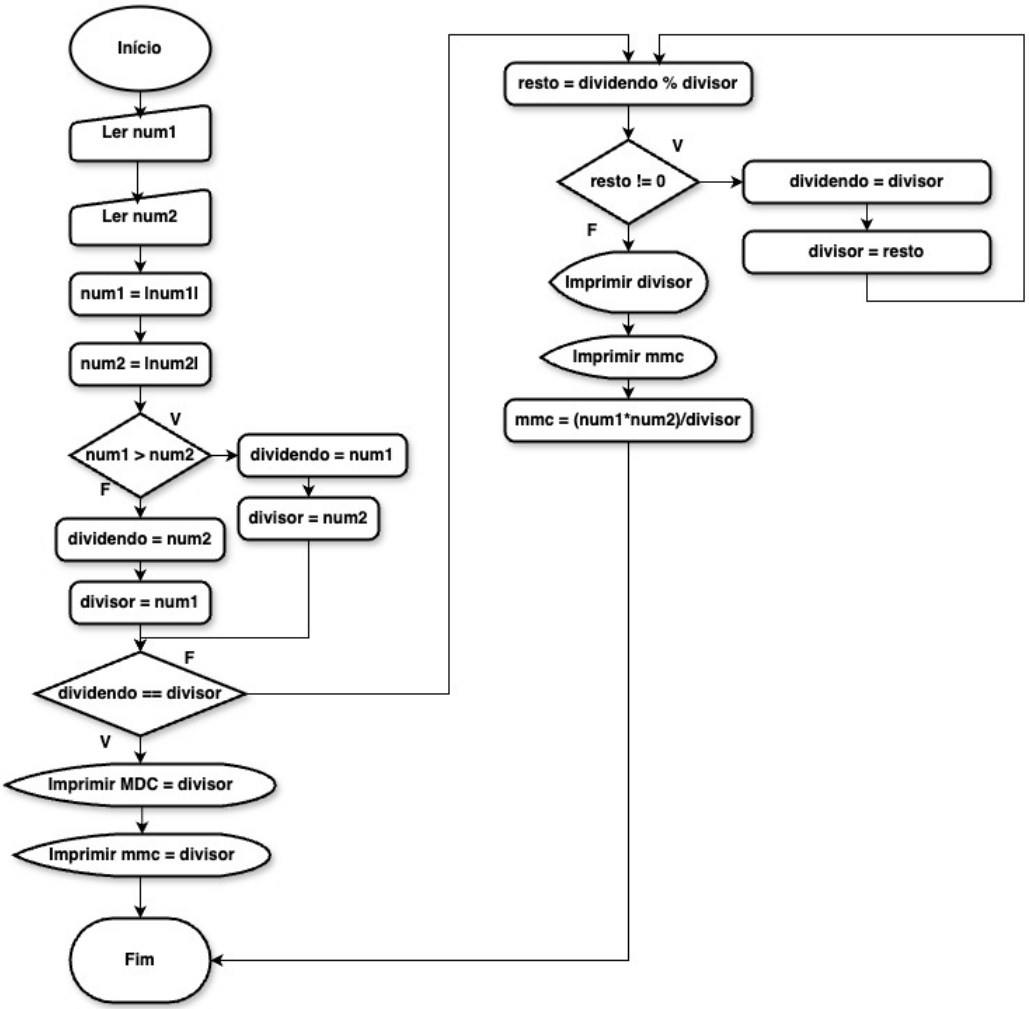



*Figura 69 – Algoritmo de cálculo do Máximo Divisor Comum e do mínimo múltiplo comum*

Para obter o MDC o algoritmo passa por obter o resto da divisão entre um dividendo e o divisor. Enquanto o resto não for igual a zero, o algoritmo continua a obter o resto, atualizando o dividendo para o último divisor, e o divisor para o último resto. Quanto o resto for igual a zero, o último divisor é o MDC. O mmc resulta do cálculo por fórmula.

```
num1 = int(input("Insira o primeiro número : "))
num2 = int(input("Insira o segundo número : "))

num1=abs(num1)
num2=abs(num2)

if num1>num2:
    dividendo=num1
    divisor=num2
else:
    dividendo=num2
    divisor=num1

if dividendo==divisor:
    print("MDC = "+str(divisor))
    print("mmc = "+str(divisor))
else:
    while True:
        resto=dividendo%divisor
        if resto!=0:
            dividendo=divisor
            divisor=resto
        else:
            break

    print("MDC = "+str(divisor))
    mmc=int((num1*num2)/divisor)
    print("mmc = "+str(mmc))
```

*Figura 70 – Programa de cálculo do Máximo Divisor Comum e do mínimo múltiplo comum*

### Número Primo

De acordo com a definição, um número primo [19] é qualquer número natural superior a um, que é apenas divisível por ele próprio e pela unidade. Como exemplos temos 2, 3, 5, 7, 11, 13, etc.. Pretende-se agora determinar se um número inserido pelo utilizador é ou não primo. Algumas verificações que poderemos ter em atenção a este algoritmo são as seguintes: números inferiores ou iguais a um não são primos; números dois e três são primos; todos os restantes números pares não são primos; todos os restantes números, ímpares, testamos a sua divisibilidade com divisores ímpares.

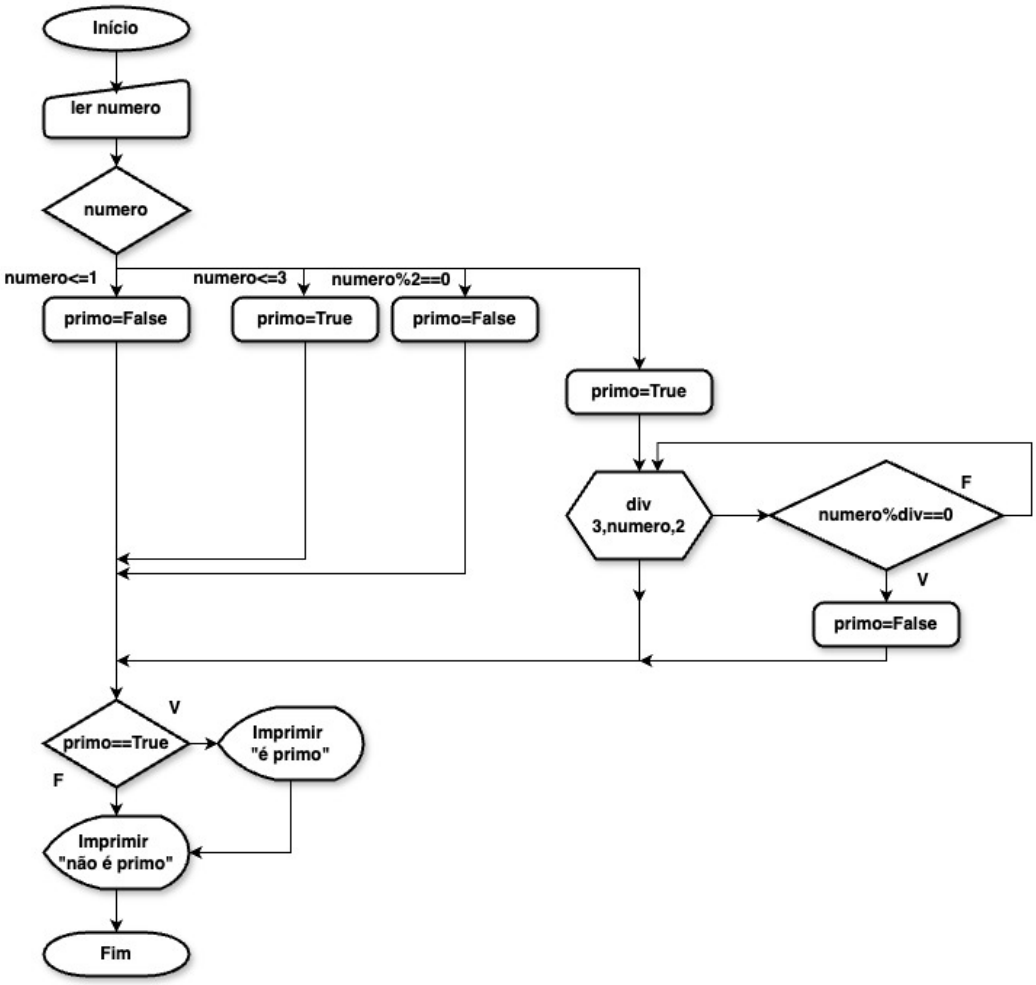



*Figura 71 - Algoritmo de teste à primalidade de um número*

```
numero = int(input("Indique número a testar : "))

if (numero<=1):
    primo=False
elif (numero <= 3):
    primo=True
elif (numero%2==0):
    primo=False
else:
    primo=True
    for div in range(3,numero,2):
        if numero % div ==0:
            primo=False
            break

if primo==True:
    print("É primo!")
else:
    print("Não é primo!")
```

*Figura 72 - Programa de teste à primalidade de um número*

### Lenda do tabuleiro de Xadrez

Apesar de existirem diversas variações ou origens geográficas da lenda, reza a história, que um súbdito chamado Sissa criou o xadrez para demonstrar o valor da estratégia e da inteligência ao rei. Encantado com o jogo, o monarca quis recompensar o inventor e ofereceu-lhe qualquer riqueza ou luxo que desejasse. Sissa pediu algo aparentemente simples: um grão de trigo para a primeira casa do tabuleiro, dois para a segunda, quatro para a terceira, e assim por diante, dobrando a quantidade até alcançar a 64ª casa. O rei, achando o pedido modesto, aceitou prontamente[20]. A quantidade de grãos de arroz será correspondente a $2^{64}$-1 grãos de arroz, ou seja, seria necessária uma área equivalente à 2.471 vezes o tamanho da Índia para que o rei pudesse cultivar o arroz necessário para pagar o inventor. A lenda pretende refletir sobre diversas questões éticas e morais. No nosso caso, algorítmicas!

O desafio que propomos é o de criar um algoritmo para calcular o total de grãos de arroz a oferecer ao súbdito. Sabemos que um tabuleiro tem 64 casas, a primeira casa terá um grão, as restantes sempre o dobro da anterior. No final pretendemos calcular o total de grãos, sabendo que a solução será $2^{64}$-1= 18446744073709551615. Este valor é extremamente grande e ilustra bem a ideia exponencial por trás da lenda.

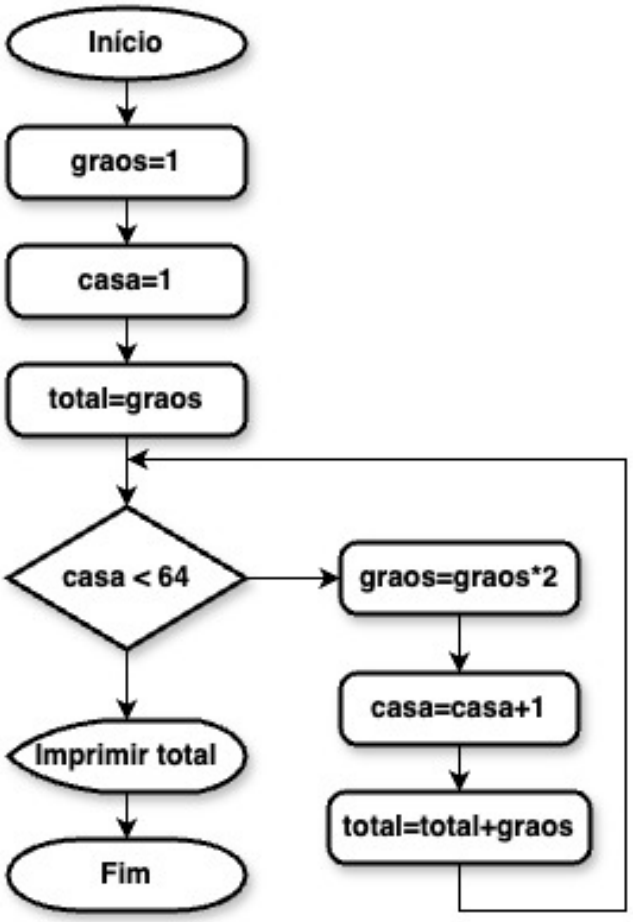



*Figura 73 - Algoritmo de cálculo de graos no tabuleiro de Xadrez*

```
graos=1
casa=1
total=graos

while casa<64:
    graos=graos*2
    casa=casa+1
    total=total+graos

print(total)
```

*Figura 74 - Programa de cálculo de graos no tabuleiro de Xadrez*

O programa começa por inicializar três variáveis: `graos` com o valor `1`, representando o número de grãos na primeira casa do tabuleiro; `casa` com o valor `1`, indicando que a contagem começa na primeira casa; `total` com o valor `1`, que acumula o número total de grãos colocados no tabuleiro.

Em seguida, utiliza um ciclo `while` para calcular o número de grãos em cada `casa` e acumular o `total`. O ciclo continua enquanto a variável casa for inferior a 64, ou seja, até que todas as casas do tabuleiro sejam contabilizadas.

Dentro do ciclo, a quantidade de grãos na próxima casa é calculada duplicando o valor da variável `graos` (isto é, multiplicando por 2). A variável `casa` é incrementada em 1 para avançar para a próxima casa. O total acumulado de grãos (`total`) é atualizado somando os grãos da casa atual ao valor acumulado anteriormente.

Após o ciclo terminar (quando todas as 64 casas forem contabilizadas), o programa utiliza a função `print()` para exibir o valor total de grãos colocados no tabuleiro.

Este programa é um exemplo clássico usado para ensinar conceitos como crescimento exponencial, ciclos `while`, e manipulação de variáveis em Python. Além disso, demonstra como pequenas quantidades podem crescer rapidamente quando duplicadas repetidamente.

**Cálculo do fatorial de um número**

O fatorial de um número inteiro positivo n, denotado por n!, é o produto de todos os números inteiros positivos de 1 até n. Em outras palavras, é a multiplicação de n por todos os números menores que ele até chegar a 1. Vejamos um exemplo simples de cálculo do fatorial de 7:

$7! = 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 5040$

Crie um algoritmo e programa que calcule o fatorial de um determinado número inteiro. Tenha em atenção os casos especiais:

$0! = 1$
$1! = 1$
$2! = 2$

Também aqui encontramos um padrão repetitivo de multiplicações. Identificando-o e definindo os casos especiais teremos o seguinte algoritmo:

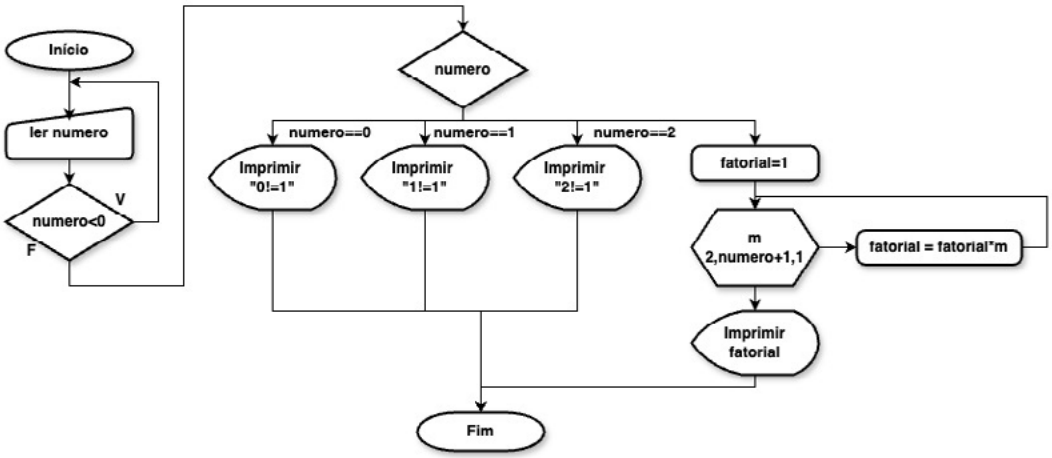



*Figura 75 - Algoritmo de cálculo do fatorial de um número*

Qualquer fatorial superior a dois, será a multiplicação do seu número, por todos os restantes números até dois (não valerá a pena multiplicar por um, apesar de o poder fazer).

```
while True:
    numero=int(input("Indique um número para cálculo do fatorial : "))
    if numero<0:
        continue
    else:
        break

if numero==0:
    print("0! = 1")
elif numero==1:
    print("1! = 1")
elif numero==2:
    print("2! = 1")
else:
    fatorial=1
    for m in range(2,numero+1,1):
        fatorial=fatorial*m
    print(str(numero)+"! = "+str(fatorial))
```

*Figura 76 - Programa de cálculo do fatorial de um número*

## Grelha de soma

Vamos agora tentar gerar a seguinte sequência de números, uma grelha de soma. A grelha de soma tem este nome porque a intersecção entre uma linha e coluna resulta na soma da linha com a coluna:

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| **1** | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| **2** | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| **3** | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
| **4** | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
| **5** | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| **6** | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
| **7** | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
| **8** | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
| **9** | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| **10** | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |

*Figura 77 - Grelha de soma*

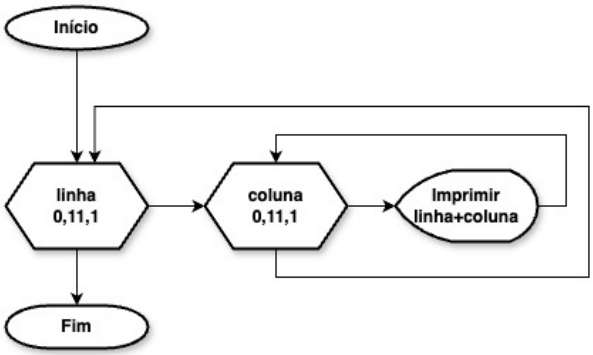



*Figura 78 - Algoritmo de geração de uma grelha de somas*

Neste algoritmos sabemos as repetições padronizadas que teremos: criação de dez linhas e a criação de dez colunas. Em cada linha, criamos as 10 colunas.

```
for linha in range(0,11,1):
    for coluna in range(0,11,1):
        print(str(linha+coluna),end=" ")

    print("\n")
```

*Figura 79 - Programa de geração de uma grelha de somas*

O programa gera e exibe uma tabela numérica onde cada célula contém a soma dos valores da linha e da coluna correspondentes. A lógica do programa utiliza dois ciclos `for` encadeados para percorrer os valores das linhas e colunas, e o resultado é apresentado de forma organizada. Segue uma descrição detalhada do funcionamento:

O programa começa com dois ciclos for encadeados. O primeiro ciclo (`for linha in range(0,11,1)`) percorre os valores das linhas, indo de `0` até `10` (inclusive). O

segundo ciclo (`for coluna in range(0,11,1)`) percorre os valores das colunas, também de `0` até `10` (inclusive). Estes ciclos permitem iterar por todas as combinações possíveis de linhas e colunas numa tabela de 11x11.

Dentro do ciclo encadeado, o programa calcula a soma dos valores da `linha` e da `coluna` atuais utilizando a expressão `linha + coluna`. Este valor é convertido para uma `string` com a função `str()` e exibido na mesma linha através da função `print()`. O argumento `end=" "` é utilizado para evitar que o cursor passe para a linha seguinte após cada valor, garantindo que os números são exibidos lado a lado na mesma linha.

Após o ciclo interno (que percorre as `colunas`) ser concluído, o programa executa um `print("\n")`, que insere uma quebra de linha. Isto faz com que os números da próxima linha sejam exibidos numa nova linha no terminal, criando o formato de tabela.

O resultado final é uma tabela onde cada célula contém a soma do número da `linha` e do número da `coluna` correspondentes. Por exemplo:

Na primeira linha (linha 0), os valores vão de 0 (0+0) até 10 (0+10).

Na segunda linha (linha 1), os valores vão de 1 (1+0) até 11 (1+10).

Este padrão continua até à última linha (linha 10), onde os valores vão de 10 (10+0) até 20 (10+10).

Este programa é útil para ilustrar conceitos básicos de ciclos encadeados e operações matemáticas em Python. Além disso, pode ser adaptado para criar outras tabelas ou matrizes com diferentes operações, intervalos numéricos e até dimensões.

### Gerador de ementas

Para encerrar a bateria de exercícios de repetições, vamos olhar para um problema que pretende gerar todas as possibilidades de ementas. Após o utilizador introduzir o número de entradas, bebidas, pratos e sobremesas, o programa deverá imprimir todas as opções possíveis e o número total. Exemplo:

```
Indique o número total de entradas : 2
Indique o número total de bebidas : 3
Indique o número total de pratos : 2
Indique o número total de sobremesas : 2
opção 1 : entrada 1, bebida 1, prato 1, sobremesa 1
opção 2 : entrada 1, bebida 1, prato 1, sobremesa 2
opção 3 : entrada 1, bebida 1, prato 2, sobremesa 1
opção 4 : entrada 1, bebida 1, prato 2, sobremesa 2
opção 5 : entrada 1, bebida 2, prato 1, sobremesa 1
opção 6 : entrada 1, bebida 2, prato 1, sobremesa 2
opção 7 : entrada 1, bebida 2, prato 2, sobremesa 1
opção 8 : entrada 1, bebida 2, prato 2, sobremesa 2
opção 9 : entrada 1, bebida 3, prato 1, sobremesa 1
opção 10 : entrada 1, bebida 3, prato 1, sobremesa 2
opção 11 : entrada 1, bebida 3, prato 2, sobremesa 1
opção 12 : entrada 1, bebida 3, prato 2, sobremesa 2
opção 13 : entrada 2, bebida 1, prato 1, sobremesa 1
opção 14 : entrada 2, bebida 1, prato 1, sobremesa 2
opção 15 : entrada 2, bebida 1, prato 2, sobremesa 1
opção 16 : entrada 2, bebida 1, prato 2, sobremesa 2
opção 17 : entrada 2, bebida 2, prato 1, sobremesa 1
opção 18 : entrada 2, bebida 2, prato 1, sobremesa 2
```

```
opção 19 : entrada 2, bebida 2, prato 2, sobremesa 1
opção 20 : entrada 2, bebida 2, prato 2, sobremesa 2
opção 21 : entrada 2, bebida 3, prato 1, sobremesa 1
opção 22 : entrada 2, bebida 3, prato 1, sobremesa 2
opção 23 : entrada 2, bebida 3, prato 2, sobremesa 1
opção 24 : entrada 2, bebida 3, prato 2, sobremesa 2
Número de opções : 24
```

Podemos abordar o problema pela geração de ciclos de repetição encadeados, ou seja, geramos o número de entradas, em cada entrada, geramos o número de bebidas, em cada bebida, geramos o número de pratos, e em cada prato, geramos o número de sobremesas. Desta forma, conseguimos gerar todas as opções possíveis:

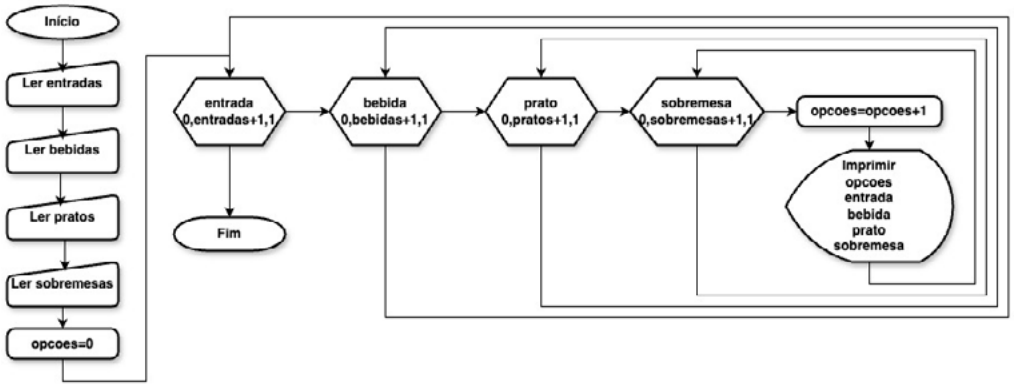



*Figura 80 - Algoritmo gerador de ementas*

```
entradas=int(input("Indique o número total de entradas : "))
bebidas=int(input("Indique o número total de bebidas : "))
pratos=int(input("Indique o número total de pratos : "))
sobremesas=int(input("Indique o número total de sobremesas : "))
opcoes=0

for entrada in range(1,entradas+1,1):
    for bebida in range(1,bebidas+1,1):
        for prato in range(1,pratos+1,1):
            for sobremesa in range(1,sobremesas+1,1):
                opcoes=opcoes+1
                print("opção "+str(opcoes)+" : entrada "+str(entrada)+
                      ", bebida "+str(bebida)+", prato "+str(prato)+
                      ", sobremesa "+str(sobremesa))

print("Número de opções : "+str(opcoes))
```

*Figura 81 - Programa gerador de ementas*

Este programa é uma simulação que calcula e apresenta todas as combinações possíveis de um menu, composto por entradas, bebidas, pratos e sobremesas. O utilizador fornece o número total de itens disponíveis em cada categoria, e o programa gera todas as opções possíveis, enumerando-as e exibindo os detalhes de cada combinação.

O programa começa por solicitar ao utilizador que insira o número total de itens disponíveis em cada categoria do menu: entradas, bebidas, pratos e sobremesas. Estes valores são introduzidos através da função `input()` e convertidos para inteiros utilizando a função `int()`. As variáveis entradas, bebidas, pratos e sobremesas armazenam os respetivos números fornecidos pelo utilizador.

Em seguida, o programa inicializa a variável `opcoes` com o valor `0`. Esta variável será utilizada para contar o número total de combinações possíveis no menu.

A lógica principal do programa utiliza quatro ciclos `for` encadeados. Cada ciclo percorre os itens de uma categoria específica (`entradas`, `bebidas`, `pratos` e `sobremesas`) utilizando intervalos definidos pela função `range()`. Por exemplo, o ciclo para as `entradas` percorre os valores de `1` até `entradas+1`, garantindo que todos os itens são considerados. O mesmo processo é aplicado às restantes categorias.

Dentro dos ciclos encadeados, o programa incrementa a variável `opcoes` em `1` para cada combinação gerada. Em seguida, utiliza a função `print()` para exibir os detalhes da combinação atual. A saída inclui o número da `opção` (por exemplo, "`opção 1`") e os itens selecionados em cada categoria (por exemplo, "`entrada 1, bebida 1, prato 1, sobremesa 1`").

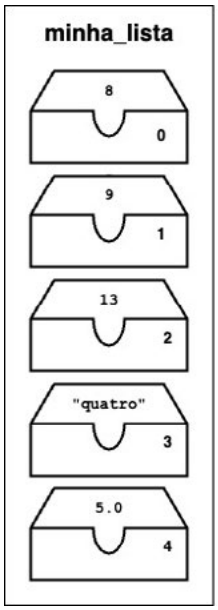
Após todos os ciclos serem concluídos, o programa apresenta o número total de combinações possíveis utilizando outra instrução `print()`. Este valor corresponde ao produto dos números fornecidos pelo utilizador para cada categoria (`entradas × bebidas × pratos × sobremesas`).

Este programa é útil para calcular todas as combinações possíveis num menu ou sistema semelhante. Pode ser aplicado em contextos como planeamento de refeições ou configuração de opções personalizáveis em sistemas de escolha.

# LISTAS

Listas em Python são estruturas de dados que permitem armazenar uma coleção de itens. Elas são mutáveis, o que significa que você pode modificar seus elementos após a criação, adicionando ou removendo itens. As listas podem conter elementos de diferentes tipos, incluindo números, strings e até outras listas.

Podemos comparar uma lista com uma variável da seguinte forma: tal como na secção Variáveis comparámos uma variável, par nome/valor que identifica um nome que guarda um determinado valor, uma lista será um nome, que guarda vários valores. Imagine, por exemplo, um arquivo de múltiplas gavetas que guarda vários valores:




```
minha_lista = [8, 9, 13, "quatro", 5.0]
```

Cada uma dessas "gavetas" é acedida por um índice(0,1,2,3,...). Por exemplo, vamos supor que pretendo imprimir o terceiro elemento da minha lista:

```
minha_lista = [8, 9, 13, "quatro", 5.0]

print(minha_lista[2])
```

```
Python 3.9.1 (v3.9.1:1e5d33e9b9, Dec  7 2020, 12:10:52)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license()" for more information.
>>>
======= RESTART: /Users/miguel/Desktop/Work/ESAN/Ensino/__book/listas.py =======
13
>>>
```

*Figura 82 – Criação de uma lista e impressão do terceiro elemento, índice 2, valor 13*

As listas em Python têm uma variedade de métodos úteis que facilitam a manipulação e gestão de seus elementos. Aqui estão alguns dos principais métodos de listas:

## Métodos Comuns

**append(x)** - Adiciona um item x ao final da lista.

```
minha_lista = [1, 2, 3]
minha_lista.append(4)  # [1, 2, 3, 4]
```

**extend(iterable)** **-** Adiciona todos os elementos de um iterável (como outra lista) ao final da lista.

```
minha_lista.extend([5, 6])  # [1, 2, 3, 4, 5, 6]
```

**insert(i, x)** - Insere um item x na posição Ωi.

```
minha_lista.insert(0, 0)  # [0, 1, 2, 3, 4, 5, 6]
```

**remove(x)** — Remove o primeiro item da lista cujo valor é igual a x. Se não houver tal item, lança um erro.

```
minha_lista.remove(3)  # [0, 1, 2, 4, 5, 6]
```

**pop([i])** - Remove e retorna o item na posição i. Se I não for especificado, remove e retorna o último item.

```
ultimo_item = minha_lista.pop()  # Remove 6
```

**clear()** - Remove todos os itens da lista.

```
minha_lista.clear()  # []
```

**index(x[, start[, end]])** - Retorna o índice do primeiro item cujo valor é igual a x, opcionalmente procurando entre os índices start e end.

```
indice = minha_lista.index(2)  # Retorna o índice do valor 2
```

**count(x)** - Retorna o número de vezes que x aparece na lista.

```
quantidade = minha_lista.count(2)  # Conta quantas vezes 2 aparece
```

**sort(key=None, reverse=False)** - Ordena os itens da lista no local (modifica a lista) em ordem ascendente por padrão. Pode-se usar key para fornecer uma função de comparação e reverse para ordenar em ordem decrescente.

```
minha_lista.sort()  # Ordena a lista
```

**reverse()** - Inverte a ordem dos itens na lista.

```
minha_lista.reverse()  # Inverte a lista
```

**copy()** - Retorna uma cópia rasa da lista.

```
copia = minha_lista.copy()  # Cria uma cópia da lista
```

### Exemplo Prático

```
minha_lista = [3, 1, 4, 1, 5]
minha_lista.append(9)
minha_lista.remove(1)
minha_lista.sort()
print(minha_lista)  # Saída: [1, 3, 4, 5, 9]
```

```
Python 3.9.1 (v3.9.1:1e5d33e9b9, Dec  7 2020, 12
:10:52)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license(
)" for more information.
>>>
====== RESTART: /Users/miguel/Desktop/Work/ESAN/
Ensino/__book/listas_2.py ======
[1, 3, 4, 5, 9]
>>>
```

*Figura 83 - Exemplo prático de aplicação de métodos de gestão de listas*

## Quiosque de votações

Vamos criar um quiosque de votação automática. Cada utilizador que aceda ao programa, pode votar na opção "A", "B" ou "C", permitindo que todos os votos fiquem guardados numa lista de nome `urna`. Quando um dos utilizadores inserir algo que não seja uma das opções, "`A`", "`B`" ou "`C`", o programa termina, calcula e apresenta o total de votos em cada opção e percentagens.

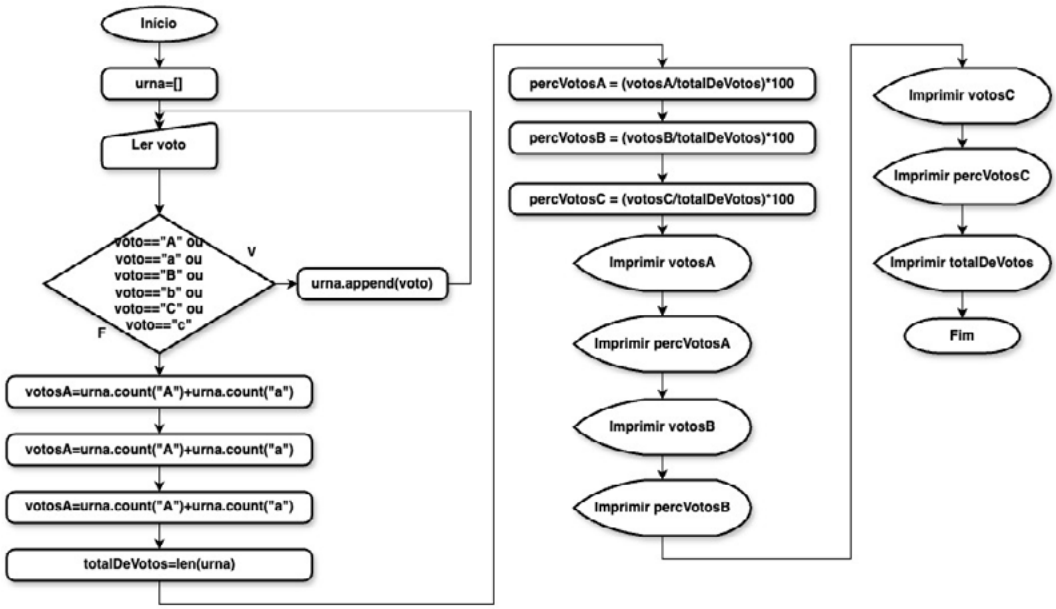



*Figura 84 - Algoritmo quiosque de votações*

```
urna=[]

while True:
    voto=input("Vote em A, B ou C : ")

    if voto=="A" or voto=="a" or voto=="B" or voto=="b" or voto=="C" or voto=="c":
        urna.append(voto)
    else:
        break

votosA=urna.count("A")+urna.count("a")
votosB=urna.count("B")+urna.count("b")
votosC=urna.count("C")+urna.count("c")

totalDeVotos = len(urna)

PercVotosA = (votosA/totalDeVotos)*100
PercVotosB = (votosB/totalDeVotos)*100
PercVotosC = (votosC/totalDeVotos)*100

print("Votos A : "+str(votosA))
print("% Votos A : "+str(PercVotosA))
print("Votos B : "+str(votosB))
print("% Votos B : "+str(PercVotosB))
print("Votos C : "+str(votosC))
print("% Votos C : "+str(PercVotosC))
print("Total de Votos : "+str(totalDeVotos))
```

*Figura 85 - Programa quiosque de votações*

Este programa em Python simula uma votação simples, onde os utilizadores podem votar numa de três opções: "A", "B" ou "C". Utiliza uma lista chamada `urna` para armazenar os votos e calcula o total de votos recebidos para cada opção, bem como a percentagem correspondente:

Primeiro, o programa cria uma lista vazia chamada `urna`, que será usada para armazenar os votos dos utilizadores. Em seguida, entra em um *loop* (ciclo) infinito (`while True`) no qual solicita ao utilizador que insira seu voto. O voto pode ser "A", "B" ou "C" (aceitando letras maiúsculas ou minúsculas). Caso o voto seja válido, ele é adicionado à lista `urna`. Se o utilizador digitar qualquer outro valor, o *loop* é interrompido, encerrando a fase de votação.

Após o término da votação, o programa conta quantos votos cada opção recebeu. Para isso, utiliza o método `count()` da lista para somar os votos de "A" e "a", de "B" e "b", e de "C" e "c". Com esses dados, calcula também o total de votos registados na urna utilizando a função `len()`.

Em seguida, o programa calcula a percentagem de votos para cada opção. Isso é feito dividindo o número de votos de cada opção pelo total de votos e multiplicando por 100. Esses valores são armazenados nas variáveis `percVotosA`, `percVotosB` e `percVotosC`.

Por fim, o programa exibe os resultados da votação, imprimindo a quantidade de votos recebidos por cada opção ("A", "B" e "C"), as respetivas percentagens e o total geral de votos registados. Esse formato permite visualizar rapidamente a distribuição dos votos entre as opções disponíveis. Este algoritmo e programa é um exemplo simples e didático que pode ser usado como base para implementar sistemas mais complexos de votação ou coleta de dados.

## Despesas de projeto

Vamos criar um programa que leia despesas para uma lista enquanto um utilizador não inserir o valor zero. Após a inserção, somamos todas as despesas, calculamos a média e apresentamos o seu total.

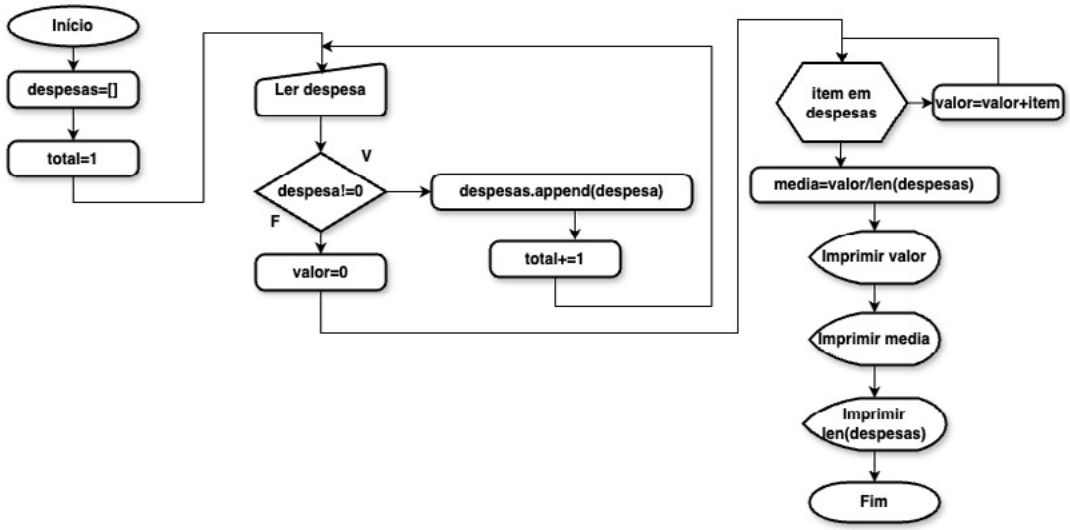



*Figura 86 – Algoritmo Despesas de Projeto*

```
despesas=[]
total=1

while True:
    despesa=float(input("Insira despesa #"+str(total)+" : "))

    if despesa!=0:
        despesas.append(despesa)
        total+=1
    else:
        break

valor=0
for item in despesas:
    valor=valor+item;

media=valor/len(despesas)

print("Total de despesas : "+str(valor))
print("Média por item : "+str(media))
print("Número de items : "+str(len(despesas)))
```

*Figura 87 - Programa Despesas de Projeto*

Este programa em Python é uma ferramenta simples para calcular o total, a média e o número de itens de uma lista de despesas introduzidas pelo utilizador. A lógica do programa está dividida em várias etapas:

O programa começa por criar uma lista vazia chamada `despesas`, que será utilizada para armazenar os valores das despesas introduzidas pelo utilizador. Também inicializa a variável `total` com o valor 1, que será usada para numerar as despesas durante a entrada de dados.

Em seguida, entra num ciclo infinito (`while True`) que solicita ao utilizador que insira o valor de uma `despesa`, utilizando a função `input()` e convertendo o valor introduzido para o tipo `float`. O número da despesa é apresentado ao utilizador através da variável `total`. Caso o valor introduzido seja diferente de `0` (indicando que ainda há despesas para registar), o programa adiciona esse valor à lista `despesas` utilizando o método `append()` e incrementa a variável `total` em `1`. Se o utilizador introduzir `0`, o ciclo é interrompido, indicando que não há mais despesas a registar.

Após a recolha das despesas, o programa calcula o total dos valores registados. Para isso, utiliza um ciclo `for` que percorre cada `item` na lista `despesas` e acumula os valores na variável `valor`. Assim, valor representa o somatório de todas as despesas introduzidas.

Com o total calculado, o programa determina a média das despesas dividindo o valor `total` (`valor`) pelo número de itens na lista (`len(despesas)`). Este cálculo é armazenado na variável `media`.

Por fim, o programa apresenta os resultados ao utilizador. Utilizando a função `print()`, exibe o total das despesas registadas (`valor`), a média por item (`media`) e o número total de itens registados (`len(despesas)`). Este programa é útil para organizar e analisar despesas de forma simples e eficiente, permitindo ao utilizador obter informações importantes sobre os seus gastos.

# FUNÇÕES

Imagine que num determinado programa pretende calcular um preço IVA repetidamente em vários valores. Ou pretende calcular o câmbio de diferentes valores. Ou converter temperaturas repetidas vezes. Poderá realizar esta tarefa, criando em cada situação a fórmula de cálculo ou as instruções de execução e processamento. Ou então, poderá criar um conjunto de instruções, sob a forma de um modelo reutilizável, para não ter de repetir código, cada vez que necessita de fazer algo similar. Para isso, poderá criar uma função!

Funções em programação são blocos de código projetados para realizar uma tarefa específica dentro de um programa. Elas podem receber parâmetros (dados de entrada), executar instruções e, em muitos casos, retornar um resultado (saída). O principal objetivo de uma função é evitar a repetição de código, permitindo que você escreva uma vez uma determinada rotina e a reutilize sempre que necessário apenas chamando o nome da função.

## Estrutura

A estrutura básica de uma função inclui:

- **Nome**: Indica o que a função faz;
- **Parâmetros** (dados de entrada): Valores que podem ser recebidos como entrada;
- **Corpo**: Bloco de instruções que serão executadas quando a função é chamada;
- **Retorno** (saída): Valor devolvido após a execução (nem toda função precisa retornar algo).

*Figura 88 – Estrutura básica de uma função*

As principais vantagens de usar funções em programação incluem:

- **Reutilização de código**: Não precisamos reescrever o mesmo código cada vez que é pretendido realizar a mesma operação;
- **Organização**: O código fica mais organizado e modular, facilitando sua compreensão e manutenção;
- **Facilidade de manutenção**: Caso uma alteração necessite ser efetuada, é realizada diretamente na função e será refletida em todos os pontos onde essa função é utilizada.

As funções podem ser nativas ou criadas pelo utilizador. Neste livro, temos utilizado diversas funções nativas, ou seja, já embutidas na linguagem de programação Python. Exemplos:

| Dados de entrada | Função | Saída |
|---|---|---|
| `numero` | `float(número)` | Converte número para decimal |
| `numero` | `int(número)` | Converte número para inteiro |
| `"texto"` | `print("texto")` | Imprime texto |
| `"mensagem"` | `input("mensagem")` | Imprime mensagem e retorna valor inserido pelo utilizador |

*Tabela 7 - Exemplos de funções nativas do Python*

O objetivo deste capítulo não é o de utilizar funções, até porque nesta fase o leitor já deverá ter alguma destreza em utilizá-las nos programas. O objetivo é que o utilizador seja capaz de criar as suas próprias funções, que permitam ser reutilizadas e, desta forma, otimizar código, poupar tempo e recursos.

A estrutura de funções em Python é simples:

- **Definição**: Utiliza-se a palavra-chave `def`, seguida pelo nome da função e parênteses (podendo conter parâmetros). O cabeçalho termina com dois pontos;
- **Corpo**: O bloco de código da função é indentado e contém as instruções que serão executadas;
- **Parâmetros**: Ou dados de entrada da função. São opcionais,colocados entre parênteses, servem para enviar dados de entrada à função;
- **Retorno**: Ou saída. O resultado pode ser devolvido usando a palavra reservada `return` (opcional).

## Exemplos

Vejamos agora alguns exemplos simples de funções e como utilizá-las:

Exemplo básico sem retorno de valor:

```
#estrutura da função
def saudacao(nome):
    print("Olá,", nome)

#programa principal

utilizador="Miguel"
saudacao(utilizador)
```

```
Python 3.9.1 (v3.9.1:1e5d33e9b9, Dec  7 2020, 12:10:52)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license()" for more information.
>>>
== RESTART: /Users/miguel/Desktop/Work/ESAN/Ensino/__book/funcoes_saudacao.py ==
Olá, Miguel
>>>
```

*Figura 89 - Utilização de uma função simples (saudacao) sem retorno de valor*

Exemplo básico com retorno de valor:

```
#estrutura da função
def soma(a, b):
    return a + b

#programa principal
num1=2
num2=3
print(soma(num1,num2))
```

```
Python 3.9.1 (v3.9.1:1e5d33e9b9, Dec  7 2020
, 12:10:52)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "lice
nse()" for more information.
>>>
==== RESTART: /Users/miguel/Desktop/Work/ESA
N/Ensino/__book/funcoes_soma.py ====
5
>>>
```

*Figura 90 - Utilização de uma função simples (soma) com retorno de valor (a+b)*

Podemos verificar no primeiro exemplo, a função `saudação(nome)` é definida sem devolver nenhum valor. Apenas imprime a palavra `Olá`, concatenada com o valor guardado na variável `nome`. É boa prática as funções estarem sempre definidas antes da execução do programa principal.

No segundo exemplo, é definida uma função de nome `soma(a,b)`, que soma os valores de duas variáveis, `a` e `b`. Neste caso, essa soma é devolvida para o programa principal com a palavra reservada `return a+b`. No programa principal, verificamos que ao invocar a função `soma(num1,num2)`, esta irá somar os valores de duas varáveis, `num1` e `num2`. Como a função `soma()` está encapsulada na função nativa `print()`, o resultado final é a impressão da soma no programa.

## Desafios

Vamos agora realizar alguns desafios. Iremos converter alguns dos programas anteriores já criados e converter processamento em função.

### Cálculo da paridade de um número

Anteriormente, já concebemos o algoritmo e o programa que verifica se um número é par. Vamos agora converter o teste de paridade numa função reutilizável:

```
def testeParidade(num):
    if num%2==0:
        print(str(num) + " é par")
    else:
        print(str(num) + " é ímpar")

numero=int(input("Insira um número: "))

testeParidade(numero)
```

```
Python 3.9.1 (v3.9.1:1e5d33e9b9, Dec  7 202
0, 12:10:52)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "lic
ense()" for more information.
>>>
===== RESTART: /Users/miguel/Desktop/Work/E
SAN/Ensino/__book/funcao_par.py =====
Insira um número: 5
5 é ímpar
>>>
```

*Figura 91 - Teste à paridade de um número em função(testeParidade)*

Neste exemplo, como apenas pretendemos informar o utilizador da paridade do número, a função não retorna valores, apenas imprime uma mensagem, indicando se o valor da variável `numero` é ou não par.

### Cálculo de ano bissexto

Vamos agora converter em função reutilizável, o algoritmo de cálculo de ano bissexto. Para tal, teremos como parâmetro de entrada o ano e neste caso, também não retornaremos nenhum valor. Apenas imprimiremos a característica do ano.

```
def bissexto(ano):
    if (ano%4==0 and ano%100!=0) or ano%400==0:
        print(str(ano)+" é bissexto!")
    else:
        print(str(ano)+" não é bissexto!")

anoDeTeste=int(input("Indique o ano : "))

bissexto(anoDeTeste)
```

```
Python 3.9.1 (v3.9.1:1e5d33e9b9, Dec  7 202
0, 12:10:52)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "lic
ense()" for more information.
>>>
=== RESTART: /Users/miguel/Desktop/Work/ESA
N/Ensino/__book/funcao_bissexto.py ==
Indique o ano : 2000
2000 é bissexto!
>>>
```

*Figura 92 - Teste para verificar se ano é bissexto(bissexto)*

Até agora testámos e executámos funções que não retornam valores. Vamos agora converter alguns programas anteriores em funções com retorno de valor.

### Máximo Divisor Comum(MDC) e Mínimo Múltiplo Comum(mmc)

Para este exercício, iremos criar duas funções. A primeira, permitirá o cálculo do MDC, recebendo dois parâmetros, dois números, e retornando o MDC. A segunda função, receberá os mesmos dois números, invocará a função de cálculo do MDC, e retornará o mmc.

```
def MDC(num1, num2):
    num1=abs(num1)
    num2=abs(num2)

    if num1>num2:
        dividendo=num1
        divisor=num2
    else:
        dividendo=num2
        divisor=num1

    if dividendo==divisor:
        return divisor
    else:
        while True:
            resto=dividendo%divisor
            if resto!=0:
                dividendo=divisor
                divisor=resto
            else:
                break

        return divisor

def mmc(num1, num2):
    num1=abs(num1)
    num2=abs(num2)

    mdc=MDC(num1,num2)
    mmc=int((num1*num2)/mdc)
    return mmc

numero1 = int(input("Insira o primeiro número : "))
numero2 = int(input("Insira o segundo número : "))

print("MDC = "+str(MDC(numero1,numero2)))
print("mmc = "+str(mmc(numero1,numero2)))
```

```
Python 3.9.1 (v3.9.1:1e5d33e9b9, Dec  7 2020, 12:10:52)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license()" for more i
nformation.
>>>
=== RESTART: /Users/miguel/Desktop/Work/ESAN/Ensino/__book/fu
ncao_MDC_mmc.py ===
Insira o primeiro número : 18
Insira o segundo número : 30
MDC = 6
mmc = 90
>>>
```

*Figura 93 - Cálculo do MDC e mmc com recurso a funções de retorno de valor*

Neste caso a função `MDC`, calcula o máximo divisor comum entre dois números (`numero1` e `numero2`) e devolve o seu valor. A função `mmc`, calcula o mínimo múltiplo comum de dois números (`numero1` e `numero2`) sendo que, a forma mais fácil de o fazer será calcular em primeira mão o máximo divisor comum. Para tal, invoca a função `MDC` já definida e realiza o cálculo. No programa principal, são invocadas as duas funções `MDC()` e `mmc()`.

### Número Primo

Para o cálculo de um número primo iremos também criar uma função de retorno de valor, com a entrada de um parâmetro - o número a calcular a primalidade.

```
def primo(numero):
    if (numero<=1):
        ePrimo=False
    elif (numero <= 3):
        ePrimo=True
    elif (numero%2==0):
        ePrimo=False
    else:
        ePrimo=True
        for div in range(3,numero,2):
            if numero % div ==0:
                ePrimo=False
                break

    return ePrimo


num = int(input("Indique número a testar : "))

if primo(num)==True:
    print("É primo!")
else:
    print("Não é primo!")
```

```
Python 3.9.1 (v3.9.1:1e5d33e9b9, Dec  7 2020, 12:10:52)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license()" for more information.
>>>
==== RESTART: /Users/miguel/Desktop/Work/ESAN/Ensino/__book/funcao_primo.py ====
Indique número a testar : 11
É primo!
>>>
==== RESTART: /Users/miguel/Desktop/Work/ESAN/Ensino/__book/funcao_primo.py ====
Indique número a testar : 51
Não é primo!
>>>
==== RESTART: /Users/miguel/Desktop/Work/ESAN/Ensino/__book/funcao_primo.py ====
Indique número a testar : 43
É primo!
>>>
```

*Figura 94 - Teste de primalidade de número com função de retorno*

Neste programa, optámos por uma função de devolve um estado de verdadeiro/falso, ou seja, se o número é (`True`) ou não é (`False`) primo. Após a execução da função `primo(numero)` no programa principal, o estado boleano devolvido é verificado para impressão de mensagem relativa à primalidade.

**Programação estruturada**

1. Correspondência entre fluxograma e programa;
2. Uso das 3 estruturas fundamentais de controlo: Sequência-Seleção-Repetição;
3. As estruturas usadas devem ter um início e um final;
4. Programa escrito com indentação (realce), espaços em branco e comentários para facilitar a leitura do mesmo;
5. Eliminação das transferências incondicionais (os gotos do Fortran);
6. Desenho descendente e segmentação em módulos (ou funções);
7. Construção de módulos (ou funções) de tamanho adequado;
8. Declaração do domínio (scope) de ação das variáveis locais (dentro das funções) e globais (o programa inteiro);
9. Documentação do programa.

# CONSIDERAÇÕES FINAIS

A escolha da linguagem Python como base para a aprendizagem de programação revela-se uma escolha assertiva pela sua simplicidade, versatilidade e ampla aceitação no mercado de trabalho contemporâneo. O percurso apresentado nos tópicos abordados — desde conceitos fundamentais de algoritmia e programação, passando por estruturas de controlo, operadores, funções e desafios práticos — proporciona uma sólida formação inicial a qualquer pessoa interessada em iniciar ou aprofundar os seus conhecimentos em desenvolvimento de software.

Ao longo do estudo dos assuntos e conteúdos organizados, torna-se evidente que a compreensão de algoritmos e estruturas de programação é essencial para criar soluções eficientes para problemas do dia a dia. Exploramos exemplos simples e práticos, como cálculos matemáticos, manipulação de listas, e construção de funções, que refletiram a aplicabilidade direta da aprendizagem em situações reais.

Destacaram-se exercícios como a implementação do clássico "Hello World", cálculo de paridade, equações matemáticas, e algoritmos de repetição sublinhando a importância do exercício prático para fixação dos conceitos. Os desafios propostos permitem não só reforçar o conteúdo, como também desenvolver raciocínio lógico, autonomia e capacidade de análise crítica, de uma forma simples e eficaz.

A base de conhecimento adquirida com este material serve não apenas para projetos educativos, mas também abre portas para áreas de aplicação como análise de dados, inteligência artificial, automação de processos, e desenvolvimento web. Aprender com exemplos concretos e desafios variados forma a confiança necessária para explorar novos projetos em Python e outras linguagens de programação. É, neste sentido, que se deseja que este livro seja também utilizado no futuro, como consulta.

Em suma, a abordagem progressiva e metódica adotada neste índice oferece um caminho claro para quem deseja ingressar no universo da programação, servindo como ponto de partida para o aperfeiçoamento contínuo e o desenvolvimento de competências cada vez mais avançadas. O domínio dos temas explorados, acompanhado de prática e curiosidade permanente, constitui o alicerce do sucesso no universo da tecnologia.

É meu desejo que tenha contribuído para adquiri competências nesta área e, desta forma, enriquecendo a sua literacia digital.

# REFERÊNCIAS BIBLIOGRÁFICAS


[1] J. L. Berggren, Episodes in the Mathematics of Medieval Islam, Nova Iorque: Springer Science+Business Media, 1986.

[2] T. Almeida, Algoritmos e Programação: Introdução, história e conceitos básicos., Universidade Federal de São Carlos - UFSCar, Sorocaba.

[3] M. A. Arbib, Brain, Machines and Mathematics, Springer-Verlag, 1987.

[4] T. A. Andreatto, Introdução à Informática e Algoritmia, Instituto Superior de Engenharia Departamento de Engenharia Informática (DEI), Instituto Politécnico do Porto.

[5] M. P. G. Fachin, The Divide-and-Conquer Method for the Solution of the Symmetric Tridiagonal Eigenproblem and Transputer Implementations, University of Kent, Canterbury, England, 1994.

[6] D. E. Knuth, The Art of Computer Programming, Volume 1: Fundamental Algorithms (3rd ed.)., Addison-Wesley, 1997.

[7] C. E. L. R. L. R. &. C. S. T. H. Cormen, Introduction to Algorithms (3rd ed.), MIT Press, 2009.

[8] D. M. R. Brian Kernighan, The C Programming Language, Pearson, 1988.

[9] J. E. C. M. Allen Downey, How to Think Like a Computer Scientist - Learning with Python, Wellesley, Massachusetts: Green Tea Press, 2002.

[10] A. Zeller, Why Programs Fail: A Guide to Systematic Debugging (2nd ed.), Morgan Kaufmann, 2009.

[11] S. Argalias, "Naming (in code) — The ultimate guide and reference".

[12] J. Przywóski, "Python Reference (The Right Way)," 2015. [Online]. Available: https://python-reference.readthedocs.io/en/latest/docs/operators/. [Acedido em 05 02 2025].

[13] L. H. Weiner, "The Roots of Structured Programming," Computer Science Group, Department of Mathematics, Texas Tech University, Lubbock, Texas 79409.

[14] G. J. Corrado Böhm, "Flow diagrams, turing machines and languages with only two formation rules," *Communications of the ACM, Volume 9, Issue 5 ,* vol. 9, nº 5, pp. 366-371, 1966.

[15] K. Chris, "Python Switch Statement — Switch Case Example," [Online]. Available: https://www.freecodecamp.org/news/python-switch-statement-switch-case-example/. [Acedido em 4 February 2025].

[16] H. S. M. Coxeter, Introduction to Geometry, John Wiley; 2nd Edition, January 1, 1969.

[17] J. Stewart, Calculus: Early Transcendentals, Cengage Learning, 2015.

[18] C. R. Cheney, A Handbook of Dates for students of British History, Cambridge University Press, 1945.

[19] H. Eves, An Introduction to the History of Mathematics, Brooks Cole, 1990.

[20] C. Pomerance, "The Search for Prime Numbers," *Scientific American,* vol. 247, nº 6, pp. 136-147 , December 1982.

[21] D. Shenk, The Immortal Game: A History of Chess, Doubleday, 2006.

# LISTA DE FIGURAS


Figura 1 — Diagrama de transformação de ingredientes culinários para receita de um bolo 14

Figura 2 - Diagrama de transformação de ingredientes culinários com instruções para confeção de um bolo 14

Figura 3 — O trio Problema, Algoritmo, Solução(Programa) 15

Figura 4 — Simbologia para a criação de fluxogramas 18

Figura 5 — Fluxograma do algoritmo e programa "Hello World!" 18

Figura 6 — Execução de um programa por um interpretador [9] 21

Figura 7 - Execução de um programa por um compilador [9] 21

Figura 8 — Erro de sintaxe no IDLE (parêntesis em falta) 22

Figura 9 — Erro de execução no IDLE (divisão impossível por zero) 23

Figura 10 — Exemplo de instrução de criação de uma variável 24

Figura 11 — Exemplos de tipos de variáveis 24

Figura 12 — Criação de variáveis no IDLE 25

Figura 13- Instrução de input 26

Figura 14- Instrução de output 26

Figura 15 - Execução do programa com input e output 26

Figura 16 — Fluxograma do algoritmo e o programa do nome e idade 27

Figura 17 - Programa do nome e idade no IDLE 27

Figura 18 - Execução do programa do cálculo do IVA sob um artigo 27

Figura 19 — Fluxograma e programa em Python para o cálculo de um preço com taxa de IVA 28

Figura 20 — Exemplo de código Python com operadores aritméticos 30

Figura 21 — Exemplo de código Python com operadores de atribuição 31

Figura 22 — Exemplo em código Python de operadores de comparação 32

Figura 23 — Estruturas de controlo em programação 33

Figura 24 — Estrutura de controlo Sequência 34

Figura 25 — Exemplo com código Python de uma estrutura sequência 34

Figura 26 — Estrutura de seleção de uma via 35

Figura 27 - Algoritmo de apresentação de preço de menu 35

Figura 28 — Programa Python do preço do menu e execução e execução 36

Figura 29 - Erro de falta de indentação não expectável 36

Figura 30 - Estrutura de seleção de duas vias 36

Figura 31 - Algoritmo de teste à paridade de número 37

Figura 32 - Programa Python do teste de paridade de um número 37

Figura 33 - Algoritmo de teste ao sinal de um número 38

Figura 34- Programa de teste de sinal de um número 38

Figura 35 - Estrutura de seleção de n vias 39

Figura 36 - Algoritmo de teste ao sinal de um número, com seleção de 3 vias 39

Figura 37 - Programa de teste de sinal de um número, com seleção de 3 vias 39

Figura 38 - Algoritmo para determinar maior e menor de dois números 40

Figura 39 - Programa para determinar maior e menor de dois números 40
Figura 40 - Algoritmo para determinar maior e menor de três números 41
Figura 41 — Programa para determinar maior e menor de três números 42
Figura 42 - Gráfico da reta y=2x+3 42
Figura 43 - Gráfico da reta y=-5x+4 43
Figura 44 - Gráfico da reta y=4 43
Figura 45 - Algoritmo do cálculo do zero de uma reta 44
Figura 46 - Programa para determinar o zero de uma reta 44
Figura 47 — Gráfico da parábola $y=x^2+2x+1$ 45
Figura 48- Gráfico da parábola $y=-x^2+6x+1$ 45
Figura 49 - Gráfico da parábola $y=4x^2+x+40$ 45
Figura 50 - Algoritmo de cálculo dos zeros de uma parábola 46
Figura 51 - Programa de cálculo dos zeros de uma parábola 47
Figura 52 - Algoritmo de uma calculadora aritmética simples 48
Figura 53 - Programa de uma calculadora científica 48
Figura 54 - Algoritmo de cálculo do IMC e classificação 49
Figura 55 - Programa de cálculo do IMC e classificação 50
Figura 56 - Algoritmo de cálculo de ano bissexto 50
Figura 57 - Programa de cálculo de ano bissexto 51
Figura 58 — Estrutura de repetição com teste à cabeça 51
Figura 59 - Algoritmo de cálculo de anos e alturas 52
Figura 60 - Programa de cálculo de alturas e anos 52
Figura 61 - Estrutura de repetição com teste à cauda 53
Figura 62 — Algoritmo de apresentação da Série de Fibonacci até um limite 54
Figura 63 - Programa de apresentação da Série de Fibonacci até um limite 54
Figura 64 — Exemplo demonstrativo de construção da repetição com teste à cabeça (algoritmo da Figura 58) 54
Figura 65 - Exemplo demonstrativo de construção da repetição com teste à cauda ( algoritmo da Figura 61) 55
Figura 66 - Estrutura de repetição com número pré-definido de ciclos 55
Figura 67 — Algoritmo de geração da tabuada de um número, definido pelo utilizador 56
Figura 68 — Programa de geração da tabuada de um número, definido pelo utilizador 56
Figura 69 — Algoritmo de cálculo do Máximo Divisor Comum e do mínimo múltiplo comum 57
Figura 70 — Programa de cálculo do Máximo Divisor Comum e do mínimo múltiplo comum 58
Figura 71 - Algoritmo de teste à primalidade de um número 58
Figura 72 - Programa de teste à primalidade de um número 59
Figura 73 - Algoritmo de cálculo de graos no tabuleiro de Xadrez 59
Figura 74 - Programa de cálculo de graos no tabuleiro de Xadrez 60
Figura 75 - Algoritmo de cálculo do fatorial de um número 61
Figura 76 - Programa de cálculo do fatorial de um número 61
Figura 77 - Grelha de soma 62
Figura 78 - Algoritmo de geração de uma grelha de somas 62

Figura 79 - Programa de geração de uma grelha de somas 62
Figura 80 - Algoritmo gerador de ementas 64
Figura 81 - Programa gerador de ementas 64
Figura 82 — Criação de uma lista e impressão do terceiro elemento, índice 2, valor 13 66
Figura 83 - Exemplo prático de aplicação de métodos de gestão de listas 68
Figura 84 - Algoritmo quiosque de votações 68
Figura 85 - Programa quiosque de votações 69
Figura 86 — Algoritmo Despesas de Projeto 70
Figura 87 - Programa Despesas de Projeto 70
Figura 88 — Estrutura básica de uma função 72
Figura 89 - Utilização de uma função simples (saudacao) sem retorno de valor 73
Figura 90 - Utilização de uma função simples (soma) com retorno de valor (a+b) 74
Figura 91 - Teste à paridade de um número em função(testeParidade) 74
Figura 92 - Teste para verificar se ano é bissexto(bissexto) 75
Figura 93 - Cálculo do MDC e mmc com recurso a funções de retorno de valor 75
Figura 94 - Teste de primalidade de número com função de retorno 76
76

# TABELAS

Tabela 1 - Operadores aritméticos 29
Tabela 2 - Operadores de atribuição 30
Tabela 3 - Operadores de comparação 31
Tabela 4 - Operadores lógicos 32
Tabela 5 — Classificação do peso em função de intervalos de valor IMC 49
Tabela 6 — Iterações de cálculo de alturas e anos do João e do Pedro 52
Tabela 7 - Exemplos de funções nativas do Python 73